Practical domain-specific debuggers using the Moldable Debugger framework

Andrei Chiş, Marcus Denker, Tudor Gîrba, Oscar Nierstrasz

Practical domain-specific debuggers using the Moldable Debugger framework[1]

Andrei Chiş[a,*], Marcus Denker[c], Tudor Gîrba[b], Oscar Nierstrasz[a]

[a] *Software Composition Group, University of Bern, Switzerland*
[b] *tudorgirba.com*
[c] *RMoD, INRIA Lille - Nord Europe, France*

Abstract

Understanding the run-time behaviour of software systems can be a challenging activity. Debuggers are an essential category of tools used for this purpose as they give developers direct access to the running systems. Nevertheless, traditional debuggers rely on generic mechanisms to introspect and interact with the running systems, while developers reason about and formulate domain-specific questions using concepts and abstractions from their application domains. This mismatch creates an abstraction gap between the debugging needs and the debugging support leading to an inefficient and error-prone debugging effort, as developers need to recover concrete domain concepts using generic mechanisms. To reduce this gap, and increase the efficiency of the debugging process, we propose a framework for developing domain-specific debuggers, called the *Moldable Debugger*, that enables debugging at the level of the application domain. The Moldable Debugger is adapted to a domain by creating and combining *domain-specific debugging operations* with *domain-specific debugging views*, and adapts itself to a domain by selecting, at run time, appropriate debugging operations and views. To ensure the proposed model has practical applicability (*i.e.*, can be used in practice to build real debuggers), we discuss, from both a performance and usability point of view, three implementation strategies. We further motivate the need for domain-specific debugging, identify a set of key requirements and show how our approach improves debugging by adapting the debugger to several domains.

Keywords: debugging, customization, domain-specific tools, user interfaces, programming environments, Smalltalk

1. Introduction

Debugging is an integral activity of the software development process, consisting in localizing, understanding, and fixing software bugs, with the goal of making software systems behave as expected. Nevertheless, despite its importance, debugging is a laborious, costly and time-consuming activity. Together with testing, debugging can take a significant part of the effort required to ensure the correct functioning of a software system [1]. Using inadequate infrastructures for performing these activities can further increase this effort [2].

[1] This work is an extended version of a previous work: The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers. In: Proceedings of 2014 7th International Conference on Software Language Engineering (SLE 2014), http://dx.doi.org/10.1007/978-3-319-11245-9_6 © Springer International Publishing Switzerland, 2014

[☆] In Computer Languages, Systems and Structures 44() p. 89113, 2015. Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014); DOI: 10.1016/j.cl.2015.08.005

[*] Corresponding author
URL: http://scg.unibe.ch/staff/andreichis (Andrei Chiş), andrei@inf.unibe.ch (Andrei Chiş), http://marcusdenker.de (Marcus Denker), http://tudorgirba.com (Tudor Gîrba), http://scg.unibe.ch/staff/oscar (Oscar Nierstrasz)

Given the pervasiveness of debugging during software maintenance and evolution, numerous debugging techniques have been proposed (*e.g.*, remote debugging, omniscient debugging [3], post-mortem debugging [4] [5], delta debugging [6] – to name a few), each with its own constraints and benefits. These techniques rely on a wide set of tools to locate, extract and analyze data about the run-time behaviour of software systems.

Among the multitude of debugging tools, debuggers are an essential category. If loggers [7] or profilers [8] record run-time data presented to developers post-mortem, debuggers enable developers to directly observe the run-time behavior of software and elicit run-time information [9, 10]. In test-driven development the debugger is used as a development tool given that it provides direct access to the running system [11]. This makes the debugger a crucial tool in any programming environment. Nevertheless, there is an abstraction gap between the way in which developers reason about object-oriented applications, and the way in which they debug them.

On the one hand, object-oriented applications use objects to capture and express a model of the application domain. Developers reason about and formulate questions using concepts and abstractions from that domain model. This fosters program comprehension as domain concepts play an important role in software development [12, 13].

On the other hand, classical debuggers focusing on generic stack-based operations, line breakpoints, and generic user interfaces do not allow developers to rely on domain concepts when debugging object-oriented applications. Furthermore, classical debuggers are less useful when the root of the cause of a bug is far away from its manifestation [14]. Raising the level of abstraction of a debugger by offering object-oriented debugging idioms [15] solves only part of the problem, as these debuggers cannot capture domain concepts constructed on top of object-oriented programming idioms. Other approaches raise the level of abstraction in different ways: back-in-time debugging, for example, allows one to inspect previous program states and step backwards in the control flow [16].

Not offering a one-to-one mapping between developer questions and debugging support forces developers to refine their high-level questions into low-level ones and mentally piece together information from various sources. For example, when developing a parser, one common action is to step through the execution until parsing reaches a certain position in the input stream. However, as it has no knowledge of parsing and stream manipulation, a generic debugger requires developers to manipulate low-level concepts like message sends or variable accesses. This abstraction gap leads to an ineffective and error-prone effort [17].

Creating a debugger that works at the level of abstraction of a given object-oriented application can eliminate the abstraction gap. This can be achieved by:

- automatically generating a debugger based on a meta-model of the application;
- providing developers with a toolset for constructing debuggers for their application.

Automatically generating a debugger requires a specification of the meta-model. The challenge for this approach is to have a meta-model at the right level of detail. Too many or too few details can lead to debuggers with the wrong sets of features. Enabling developers to construct debuggers can lead to debuggers that have only the necessary features. The challenge here is for the toolset to make it possible to create expressive and performant debuggers with low effort.

The first approach has been applied successfully for language workbenches where the domain model of an application is expressed using external domain-specific languages that have a grammar definition [18, 19, 20], as well as in the case of domain-specific modelling languages that have an explicit specification of the meta-model [21]. Nevertheless, object-oriented programming already has a meta-model in terms of objects and their interactions: object-oriented applications provide an instantiation of the meta-model that expresses domain abstractions through concrete objects and object interactions [22]. To improve development and evolution, these object models take advantage of internal DSLs [23] (*e.g.*, APIs) instead of encoding domain concepts through external DSLs that require a grammar specification.

One can provide an additional specification of an object model from which a debugger can be generated. Nevertheless, non-trivial object-oriented applications contain rich object models [24], which can introduce significant overhead in creating and maintaining an extra specification. One can also argue that important concepts from object-oriented applications should be encoded using external DSLs that have an explicit

grammar (*i.e.*, meta-model). This, however, does not acknowledge that object-oriented programming already provides the meta-model. In this paper we investigate an alternative solution: we propose a framework for enabling developers to create domain-specific debuggers for their object-oriented applications directly on the existing object model.

When looking at a debugger, there exist two main approaches to address, at the application level, the gap between the debugging needs and debugging support:

- enable developers to create domain-specific debugging operations for stepping through the execution, setting breakpoints, checking invariants [25, 26, 27] and querying stack-related information [28, 29, 30, 31];
- provide debuggers with domain-specific user interfaces that do not necessarily have a predefined content or a fixed layout [32].

Each of these directions addresses individual debugging problems (*i.e.*, interacting with the runtime at the right level of abstraction and displaying data relevant for the application domain), however until now there does not exist one comprehensive approach to tackle the overall debugging puzzle. We propose an approach that incorporates both of these directions in one coherent model. We start from the realization that the most basic feature of a debugger model is to enable the customization of all aspects, and we design a debugging model around this principle. We call our approach the *Moldable Debugger*.

The Moldable Debugger decomposes a domain-specific debugger into a *domain-specific extension* and an *activation predicate*. The domain-specific extension customizes the user interface and the operations of the debugger, while the *activation predicate* captures the state of the running program in which that domain-specific extension is applicable. In a nutshell, the Moldable Debugger model allows developers to *mold* the functionality of the debugger to their own domains by creating domain-specific extensions. Then, at run time, the Moldable Debugger adapts to the current domain by using activation predicates to select appropriate extensions.

A domain-specific extension consists of *(i)* a set of domain-specific *debugging operations* and *(ii)* a domain-specific *debugging view*, both built on top of *(iii)* a *debugging session*. The *debugging session* abstracts the low-level details of a domain. *Domain-specific operations* reify debugging operations as objects that control the execution of a program by creating and combining *debugging events*. We model debugging events as objects that encapsulate *a predicate over the state of the running program* (*e.g.*, method call, attribute mutation) [33]. A *domain-specific debugging view* consists of a set of graphical widgets that offer debugging information. Each widget locates and loads, at run-time, relevant domain-specific operations using an annotation-based approach.

To validate our model and show that it has practical applicability, we implemented it in Pharo [34], a modern Smalltalk environment and use it to create multiple real-world debuggers. The Moldable Debugger implementation is written in less than 2000 lines of code. We have instantiated it for several distinct domains and each time the implementation required between 200-600 lines of code. We consider that its small size makes it easy to understand, and makes the adaptation of the debugger to specific domains an affordable activity. We further explore three approaches for controlling the execution of the debugged program, approaches applicable depending on the particular aspects of the target domain.

This article extends our previous work [35] as follows: *(i)* we present a more in-depth description of the Moldable Debugger model, *(ii)* we introduce two new examples of domain-specific debuggers created using the Moldable Debugger model, *(iii)* we provide a thorough discussion of related work, and *(iv)* we discuss three approaches for implementing the Moldable Debugger model.

The overall contributions of this paper are as follows:

- Identifying and discussing requirements that an infrastructure for developing domain-specific debuggers should support;
- Discussing the Moldable Debugger, a model for creating and working with domain-specific debuggers that integrates domain-specific debugging operations with domain-specific user interfaces;

- Examples illustrating the advantages of the Moldable Debugger model over generic debuggers;
- A prototype implementation of the Moldable Debugger model together with a discussion of three different approaches for implementing domain-specific debugging operations.

2. Motivation

Debuggers are comprehension tools. Despite their importance, most debuggers only provide low-level operations that do not capture user intent and standard user interfaces that only display generic information. These issues can be addressed if developers are able to create domain-specific debuggers adapted to their domain concepts. Domain-specific debuggers can provide features at a higher level of abstraction that match the domain model of software applications.

In this section we establish and motivate four requirements that an infrastructure for developing domain-specific debuggers should support, namely: *domain-specific user interfaces*, *domain-specific debugging operations*, *automatic discovery* and *dynamic switching*.

2.1. Domain-specific user interfaces

User interfaces of software development tools tend to provide large quantities of information, especially as the size of systems increases. This, in turn, increases the navigation effort of identifying the information relevant for a given task. While some of this effort is unavoidable, part of it is simply overhead caused by how information is organized on screen [36].

Consider a unit test with a failing equality assertion. In this case, the only information required by the developer is the difference between the expected and the actual value. However, finding the exact difference in non-trivial values can be daunting and can require multiple interactions such as finding the place in the stack where both variables are accessible, and opening separate inspectors for each values. A better approach, if a developer opens a debugger when a test fails, is to show a diff view on the two values directly in the debugger when such an assertion exception occurs, without requiring any further action.

This shows that user interfaces that extract and highlight domain-specific information have the power to reduce the overall effort of code understanding [37]. However, today's debuggers tend to provide generic user interfaces that cannot emphasize what is important in application domains. To address this concern, an infrastructure for developing domain-specific debuggers should:

- allow domain-specific debuggers to have *domain-specific user interfaces* displaying information relevant for their particular domains;
- support the *fast prototyping* of domain-specific user interfaces for debugging.

While other approaches, like *deet* [38] and *Debugger Canvas* [32], support domain-specific user interfaces for different domains, they do not offer an easy and rapid way to develop such domain-specific user interfaces.

2.2. Domain-specific debugging operations

Debugging can be a laborious activity requiring much manual and repetitive work. On the one hand, debuggers support language-level operations, while developers think in terms of domain abstractions. As a consequence, developers need to mentally construct high-level abstractions on top of language constructs, which can be time-consuming. On the other hand, debuggers rarely provide support for identifying and navigating through those high-level abstractions. This leads to repetitive tasks that increase debugging time.

Consider a framework for synchronous message passing. One common use case in applications using it is the delivery of a message to a list of subscribers. When debugging this use case, a developer might need to *step to when the current message is delivered to the next subscriber*. One solution is to manually step through the execution until the desired code location is reached. Another consists in identifying the code location beforehand, setting a breakpoint there and resuming execution. In both cases the developer has to manually perform a series of actions each time she wants to execute this high-level operation.

A predefined set of debugging operations cannot anticipate and capture all relevant situations. Furthermore, depending on the domain different debugging operations are of interest. Thus, an infrastructure for developing domain-specific debuggers should:

- Support the creation of domain-specific debugging operations that allow developers to *express and automate* high-level abstractions from application domains (*e.g.*, creating domain-specific breakpoints, building and checking invariants, altering the state of the running system). Since developers view debugging as an event-oriented process, the underlying mechanism should allow developers to treat the running program as a generator of events, where an event corresponds to the occurrence of a particular action during the program's execution, like: method entry, attribute access, attribute write or memory access.
- Group together those debugging operations that are relevant for a domain and only make them available to developers when they encounter that domain.

This idea of having *customizable* or *programmable* debugging operations that view debugging as an event-oriented activity has been supported in related works [25, 26, 27, 38]. Mainstream debuggers like GDB have, to some extent, also incorporated it. We also consider that debugging operations should be grouped based on the domain and only usable when working with that domain.

2.3. Automatic discovery

Based on an observational study of 28 professional developers *Roehm et al.* report that none of them used a dedicated program comprehension tool; some were not aware of standard features provided by their IDE [9]. Another study revealed that despite their usefulness and long lasting presence in IDEs, refactoring tools are heavily underused [39].

In the same way, *developers need help to discover domain-specific debuggers during debugging.* For example, if while stepping through the execution of a program a developer reaches a parser, the environment should facilitate the discovery of a domain-specific debugger that can be used in that context; if later the execution of the parser completes and the program continues with the propagation of an event, the environment should inform the developer that the current domain-specific debugger is no longer useful and that a better one exists. This way, the burden of finding appropriate domain-specific debuggers and determining when they are applicable does not fall on developers.

Recommender systems typically address the problem of locating useful software tools/commands by recording and mining usage histories of software tools [40] (*i.e.*, what tools developers used as well as how they used them). This requires, at least, some usage history information. To eliminate this need an infrastructure for developing domain-specific debuggers should *allow each domain-specific debugger to encapsulate the situations/domains in which it is applicable.*

2.4. Dynamic switching

Even with just two different types of debuggers, *DeLine et al.* noticed that users needed to switch between them at run time [32]. This happened as users did not know in advance in what situation they would find themselves in during debugging. Thus, they often did not start with the appropriate one.

Furthermore, even if one starts with the right domain-specific debugger, during debugging situations can arise requiring a different one. For example, the following scenario can occur: *(i)* while investigating how an event is propagated through the application *(ii)* a developer discovers that it is used to trigger a script constructing a GUI, and later learns that *(iii)* the script uses a parser to read the content of a file and populate the GUI. At each step a different domain-specific debugger can be used. For this to be feasible, *domain-specific debuggers should be switchable at debug time without having to restart the application.*

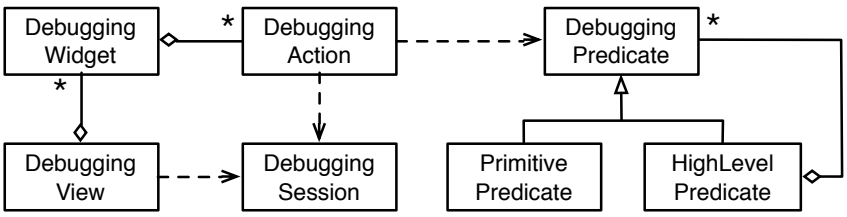

Figure 1: The structure of a domain-specific extension.

2.5. Summary

Generic debuggers focusing on low-level programming constructs, while universally applicable, cannot efficiently answer domain-specific questions, as they make it difficult for developers to take advantage of domain concepts. Domain-specific debuggers aware of the application domain can provide direct answers. We advocate that a debugging infrastructure for developing domain-specific debuggers should support the four aforementioned requirements (*domain-specific user interfaces*, *domain-specific debugging operations*, *automatic discovery* and *dynamic switching*).

3. A closer look at the "Moldable Debugger" model

The Moldable Debugger explicitly supports domain-specific debuggers that can express and answer questions at the application level. A domain-specific debugger consists of *a domain-specific extension* encapsulating the functionality and *an activation predicate* encapsulating the situations in which the extension is applicable. This model makes it possible for multiple domain-specific debuggers to coexist at the same time.

To exemplify the ideas behind the proposed solution we will instantiate a domain-specific debugger for working with synchronous events[2]. Event-based programming poses debugging challenges as it favors a control flow based on events not supported well by conventional stack-based debuggers.

3.1. Modeling domain-specific extensions

A domain-specific extension defines the functionality of a domain-specific debugger using multiple *debugging operations* and a *debugging view*. Debugging operations rely on *debugging predicates* to implement high-level abstractions (*e.g.*, domain-specific breakpoints); the debugging view highlights contextual information. To decouple these components from the low-level details of a domain they are built on top of a *debugging session*.

A *debugging session* encapsulates the logic for working with processes and execution contexts (*i.e.*, stack frames). It further implements common stack-based operations like: *step into*, *step over*, *resume/restart process*, *etc.* Domain-specific debuggers can extend the debugging session to extract and store custom information from the runtime, or provide fine-grained debugging operations. For example, our event-based debugger extends the debugging session to extract and store the current event together with the sender of that event, the receiver of that event, and the announcer that propagated that event.

Debugging predicates detect *run-time events*. Basic run-time events (*e.g.*, method call, attribute access) are detected using a set of *primitive predicates*, detailed in Table 1. More complex run-time events are detected using *high-level predicates* that combine both *primitive predicates* and other *high-level predicates* (Figure 1). Both of these types of debugging predicates are modeled as objects whose state does not change after creation. Debugging predicates are related to *coupling invariants* from data refinement, as coupling invariants are traditionally defined as logical formulas that relate concrete variables to abstract variables [41]. Hence, they can detect specific conditions during the execution of a program.

Consider our event-based debugger. This debugger can provide high-level predicates to detect when a sender initiates the delivery of an event, or when the middleware delivers the event to a receiver.

[2]This section briefly describes this debugger. More details are given in Section 4.2.

Attribute read	detects when a field of any object of a certain type is accessed
Attribute write	detects when a field of any object of a certain type is mutated
Method call	detects when a given method is called on any object of a certain type
Message send	detects when a specified method is invoked from a given method
State check	checks a generic condition on the state of the running program (*e.g.*, the identity of an object).

Table 1: Primitive debugging predicates capturing basic events.

Debugging operations can execute the program until a debugging predicate is matched or can perform an action every time a debugging predicate is matched. They are modeled as objects that can accumulate state. They can implement breakpoints, log data, watch fields, change the program's state, detect violations of invariants, *etc.* In the previous example a debugging operation can be used to stop the execution when an event is delivered to a receiver. Another debugging operation can log all events delivered to a particular receiver without stopping the execution.

At each point during the execution of a program only a single debugging operation can be active. Thus, debugging operations have to be run sequentially. For example, in the events debugger one cannot activate, at the same time, two debugging operations, each detecting when an event of a certain type is sent to a receiver. One can, however, create a single debugging operation that detects when an event of either type is sent to a receiver. This design decision simplifies the implementation of the model, given that two conflicting operations cannot run at the same time. Hence, no conflict resolution mechanism is required.

The Moldable Debugger models a *debugging view* as a collection of graphical widgets (*e.g.*, stack, code editor, object inspector) arranged using a particular layout. At run time, each widget loads a subset of debugging operations. Determining what operations are loaded by which widgets is done at run time via a lookup mechanism of operation declarations (implemented in practice using annotations). This way, widgets do not depend upon debugging operations, and are able to reload debugging operations dynamically during execution.

Our event-based debugger provides dedicated widgets that display an event together with the sender and the receiver of that event. These widgets load and display the debugging operations for working with synchronous events, like logging all events or placing a breakpoint when an event is delivered to a receiver.

Developers can create domain-specific extensions by:

(i) extending the debugging session with additional functionality;

(ii) creating domain-specific debugging predicates and operations;

(iii) specifying a domain-specific debugging view;

(iv) linking debugging operations to graphical widgets;

3.2. Combining predicates

We support two boolean operators for combining debugging predicates:

and(predicate1, predicate2, ..., predicateN): creates a new predicate that detects a run-time event when all given predicates detected a run-time event at the same time. This only allows for combining *attribute read*, *attribute write*, *method call* and *message send* with one or more *state check* predicates. For example, detecting when a method is called on an a given object is done by using a *method call* predicate together with a *state check* predicate verifying the identify of the receiver object.

or(predicate1, predicate2, ..., predicateN): creates a new predicate that detects a run-time event when either one of the given predicates detected a run-time event. For example, detecting when any message is

sent from a given method is done by using a *message send* predicate for every message send from the given method.

Given the definition of the *and* predicate, detecting high-level events that only happen when a sequence of events is detected is not possible. For example, one cannot detect, just by combing debugging predicates, a sequence of method calls on a given object. This operation requires persistent state and can be implemented by a debugging action.

3.3. Dynamic Integration

The Moldable Debugger model enables each domain-specific debugger to decide if it can handle or not a debugging situation by defining an *activation predicate*. Activation predicates capture the state of the running program in which a domain-specific debugger is applicable. While debugging predicates are applied on an execution context, activation predicates are applied on the entire execution stack. For example, the activation predicate of our event-based debugger will check if the execution stack contains an execution context involving an event.

This way, developers do not have to be aware of applicable debuggers a priori. At any point during debugging they can see what domain-specific debuggers are applicable (*i.e.*, their activation predicate matches the current debugging context) and can switch to any of them.

When a domain-specific debugger is no longer appropriate we do not automatically switch to another one. Instead, all domain-specific widgets and operations are disabled. This avoids confronting users with unexpected changes in the user interface if the new debugging view has a radically different layout/content. Nevertheless, for complex user interfaces where many widgets need to be disabled this solution can still lead to unexpected changes, though this is not as radical as replacing the user interface with a different one. Designing the disabled widgets in a way that does not confuse users could alleviate part of this issue (*e.g.*, by showing a grayed out version of the widget with no interaction possibilities).

To further improve working with multiple domain-specific debuggers we provide two additional concepts:

A debugger-oriented breakpoint is a breakpoint that when reached opens the domain-specific debugger best suited for the current situation. If more than one view is available the developer is asked to choose one.

Debugger-oriented steps are debugging operations that resume execution until a given domain-specific debugger is applicable. They are useful when a developer knows a domain-specific debugger will be used at some point in the future, but is not sure when or where.

4. Addressing domain-specific debugging problems

To demonstrate that the Moldable Debugger addresses the requirements identified in Section 2 we have instantiated it for six different domains: *testing*, *synchronous events*, *parsing*, *internal DSLs*, *profiling* and *bytecode interpretation*. In this section we detail these instantiations.

4.1. Testing with SUnit

SUnit is a framework for creating unit tests [42]. The framework provides an assertion to check if a computation results in an expected value. If the assertion fails the developer is presented with a debugger that can be used to compare the obtained value with the expected one. If these values are complex, identifying the difference may be time consuming. A solution is needed to *facilitate comparison*.

To address this, we developed a domain-specific debugger having the following components:

Session: extracts the expected and the obtained value from the runtime;

View: displays a diff between the textual representation of the two values. The diff view depends on the domain of the data being compared. Figure 2 shows how it can be used to compare two HTTP headers;

Activation predicate: verifies if the execution stack contains a failing equality assertion.

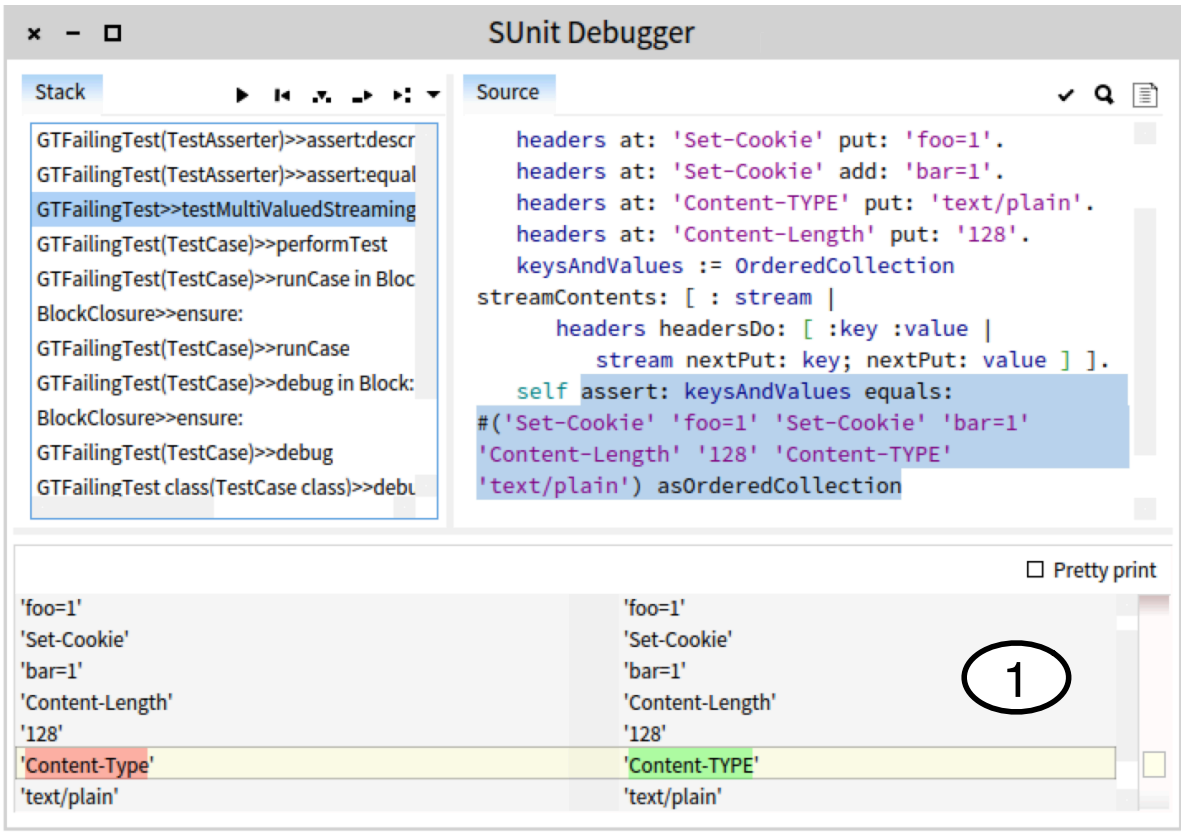

Figure 2: A domain-specific debugger for SUnit: (1) diff between the textual representation of the expected and obtained value.

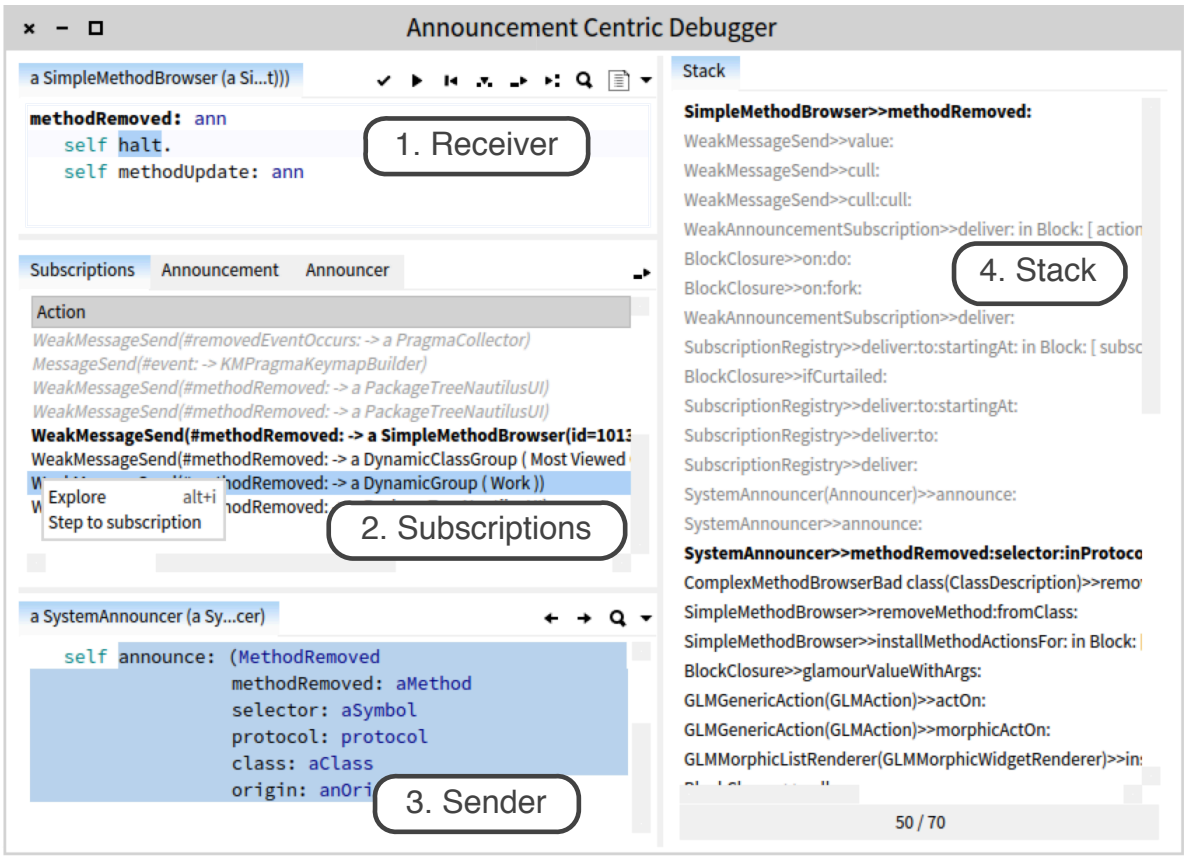

Figure 3: A domain-specific debugger for announcements: (1)(3) the receiver and the sender of an announcement; (2) subscriptions triggered by the current announcement.

4.2. An Announcement-Centric debugger

The *Announcements* framework from Pharo provides a synchronous notification mechanism between objects based on a registration mechanism and first class announcements (*i.e.*, objects storing all information relevant to particular occurrences of events). Since the control flow for announcements is event-based, it does not match well the stack-based paradigm used by conventional debuggers. For example, Section 2.2 describes a high-level action for *delivering an announcement to a list of subscribers*. Furthermore, when debugging announcements it is useful *to see at the same time both the sender and the receiver of an announcement*; most debuggers only show the receiver.

To address these problems we have created a domain-specific debugger, shown in Figure 3. A previous work discusses in more details the need for such a debugger and looks more closely at the runtime support needed to make the debugger possible [43]. This debugger is instantiated as follows:

Session: extracts from the runtime the announcement, the sender, the receiver and all the other subscriptions triggered by the current announcement;

Predicates:

Detect when the framework initiates the delivery of a subscription: *message send*(deliver: in SubscriptionRegistry >> deliver:to:startingAt:[3])

[3]We use the notation ClassName>>methodName to identify methods. For readers unfamiliar with Smalltalk code deliver:to:startingAt: is a method that takes three arguments.

Detect when the framework delivers a subscription to an object: *method call*(aSubscription action selector) on the class of the object and *state check* verifying the identity of the target object and *state check* verifying that the message was sent from the announcer holding the given subscription;

Operations:

Step to the delivery of the next subscription;

Step to the delivery of a selected subscription;

View: shows both the sender and the receiver of an announcement, together with all subscriptions served as a result of that announcement;

Activation predicate: verifies if the execution stack contains an execution context initiating the delivery of an announcement.

4.3. A debugger for PetitParser

PetitParser is a framework for creating parsers, written in Pharo, that makes it easy to dynamically reuse, compose, transform and extend grammars [44]. A parser is created by specifying a set of grammar productions in one or more dedicated classes. When a parser is instantiated the grammar productions are used to create a tree of primitive parsers (*e.g.*, choice, sequence, negation); this tree is then used to parse the input.

Whereas most parser generators instantiate a parser by generating code, PetitParser generates a dynamic graph of objects. Nevertheless, the same issues arise as with conventional parser generators: generic debuggers do not provide debugging operations at the level of the input (*e.g.*, set a breakpoint when a certain part of the input is parsed) and of the grammar (*e.g.*, set a breakpoint when a grammar production is exercised). Generic debuggers also do not display the source code of grammar productions nor do they provide easy access to the input being parsed. To overcome these issues, other tools for working with parser generators provide dedicated domain-specific debuggers. For example, ANTLR Studio an IDE for the ANTLR [45] parser generator provides both breakpoints and views at the level of the grammar [46]. Rebernak *et al.* also give an example of a dedicated debugger for ANTLR [47].

In the case of PetitParser we have developed a domain-specific debugger by configuring the Moldable Debugger as follows:

Session: extracts from the runtime the parser and the input being parsed;

Predicates:

Detect the usage of any type of parser: *method call*(parseOn:) predicates combined using *or* on all subclasses of PPParser that are not abstract and override the method parseOn:;

Detect the usage of any type of production: *method call*(PPDelegateParser>>parseOn:);

Detect the usage of a specific primitive parser: *method call*(parseOn:) predicates combined using *or* on all subclasses of PPParser that represent a primitive parser (*e.g.*, PPRepeatingParser);

Detect the usage of a specific production: *method call*(PPDelegateParser>>parseOn:) and *state check* verifying that the receiver object is a parser for the given grammar production;

Detect when a parser fails to match the input: *method call*(PPFailure class>>message:context:), or *method call*(PPFailure class>>message:context:at:);

Detect when the position of the input stream changes: *attribute write*(#position from PPStream) and *state check* verifying that the attribute value changed;

Detect when the position of the input stream reaches a given value: *attribute write*(#position from PPStream) and *state check* verifying that the attribute value is set to a given value;

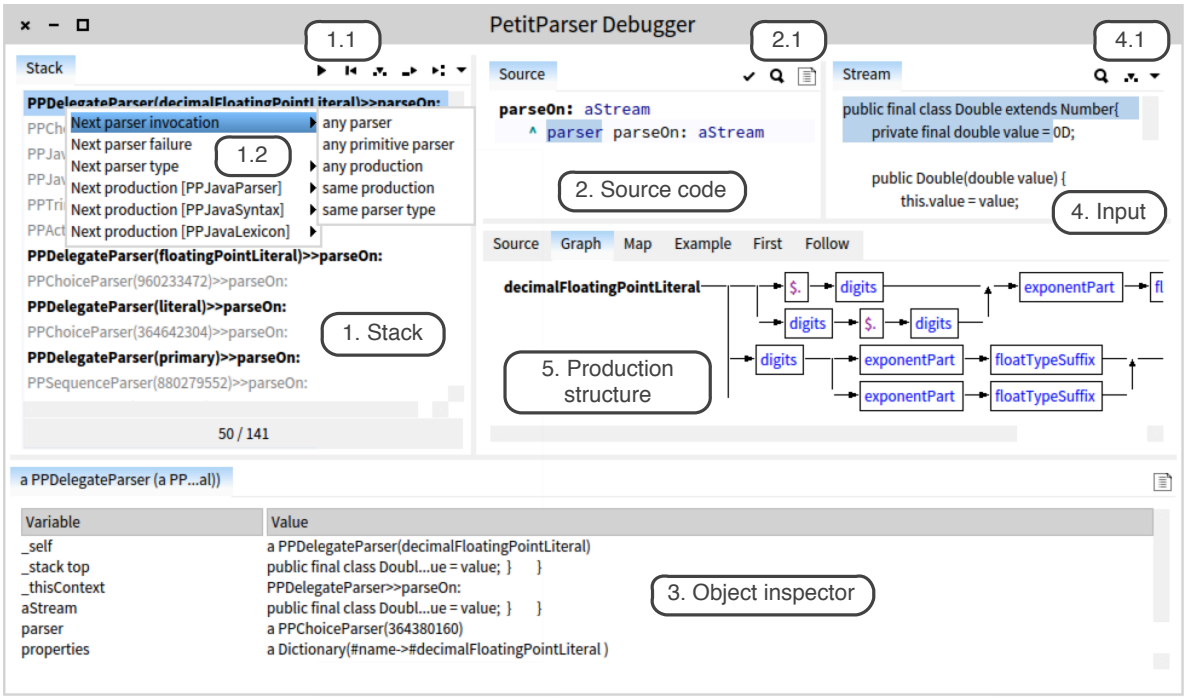

Figure 4: A domain-specific debugger for PetitParser. The debugging view displays relevant information for debugging parsers ((4) Input, (5) Production structure). Each widget loads relevant debugging operations (1.1, 1.2, 2.1, 4.1).

Operations: Navigating through the execution at a higher level of abstraction is supported through the following debugging operations:

Next parser: step until a primitive parser of any type is reached

Next production: step until a production is reached

Primitive parser(aPrimitiveParserClass): step until a parser having the given class is reached

Production(aProduction): step until the given production is reached

Next failure: step until a parser fails to match the input

Stream position change: step until the stream position changes (it either increases, if a character was parsed, or decrease if the parser backtracks)

Stream position(anInteger): step until the stream reaches a given position

View: The debugging view of the resulting debugger is shown in Figure 4. We can see that now the input being parsed is incorporated into the user interface; to know how much parsing has advanced, the portion that has already been parsed is highlighted. Tabs are used to group six widgets showing different types of data about the current production, like: source code, structure, position in the whole graph of parsers, an example that can be parsed with the production, *etc.* The structure of the parser (*e.g.*, the *Graph* view in Figure 4), for example, is generated from the object graph of a parser and can allow developers to navigate a production by clicking on it. The execution stack further highlights those execution contexts that represent a grammar production;

Activation predicate: verifies if the execution stack contains an execution context created when using a parser.

4.4. A debugger for Glamour

Glamour is an engine for scripting browsers based on a components and connectors architecture [48]. New browsers are created by using an internal domain-specific language (DSL) to specify a set of *presentations* (graphical widgets) along with a set of *transmissions* between those presentations, encoding the information flow. Users can attach various conditions to transmissions and alter the information that they propagate. Presentations and transmissions form a model that is then used to generate the actual browser.

The Moldable Debugger relies on Glamour for creating domain-specific views. Thus, during the development of the framework we created a domain-specific debugger to help us understand the creation of a browser:

Session: extracts from the runtime the model of the browser;

Predicates:

Detect the creation of a presentation:
message send(glamourValue: in GLMPresentStrategy>>presentations);

Detect when a transmission alters the value that it propagates:
message send(glamourValue: in GLMTransmission>>value);

Detect when the condition of a transmission is checked:
message send(glamourValue: in GLMTransmission>>meetsCondition);

Operations:

Step to presentation creation

Step to transmission transformation

Step to transmission condition

View: displays the structure of the model in an interactive visualization that is updated as the construction of the model advances (Figure 5);

Activation predicate: verifies if the execution stack contains an execution context that triggers the construction of a browser.

4.5. Profiler framework

Spy is a framework for building custom profilers [49]. Profiling information is obtained by executing dedicated code before or after method executions. This code is inserted into a method by replacing the target method with a new method (*i.e.*, method wrapper [50]) that executes the code of the profiler before and after calling the target method. Hence, if a developer needs to debug profiled code she has to manually skip over the code introduced by the profiler to reach the target method. To address this issue, together with the developers of S2py [51] (the second version of the Spy framework), we created a domain-specific debugger that does not expose developers to profiler code when debugging methods are being profiled:

Session: no changes in the default debugging session are required;

Predicates:

Detect when Spy finished executing profiling code for a method
message send(valueWithReceiver:arguments: in S2Method>>run:with:in:)

Operations:

Step into method call ignoring profiled code: debugging action that when stepping into a method profiling code automatically steps over the profiler code and into the code of the original method.

View: has the same widgets as the user interface of the default debugger. However, the stack widget removes all stack frames internal to the Spy framework.

Activation predicate: verifies if the execution stack contains an execution context that starts a Spy profiler.

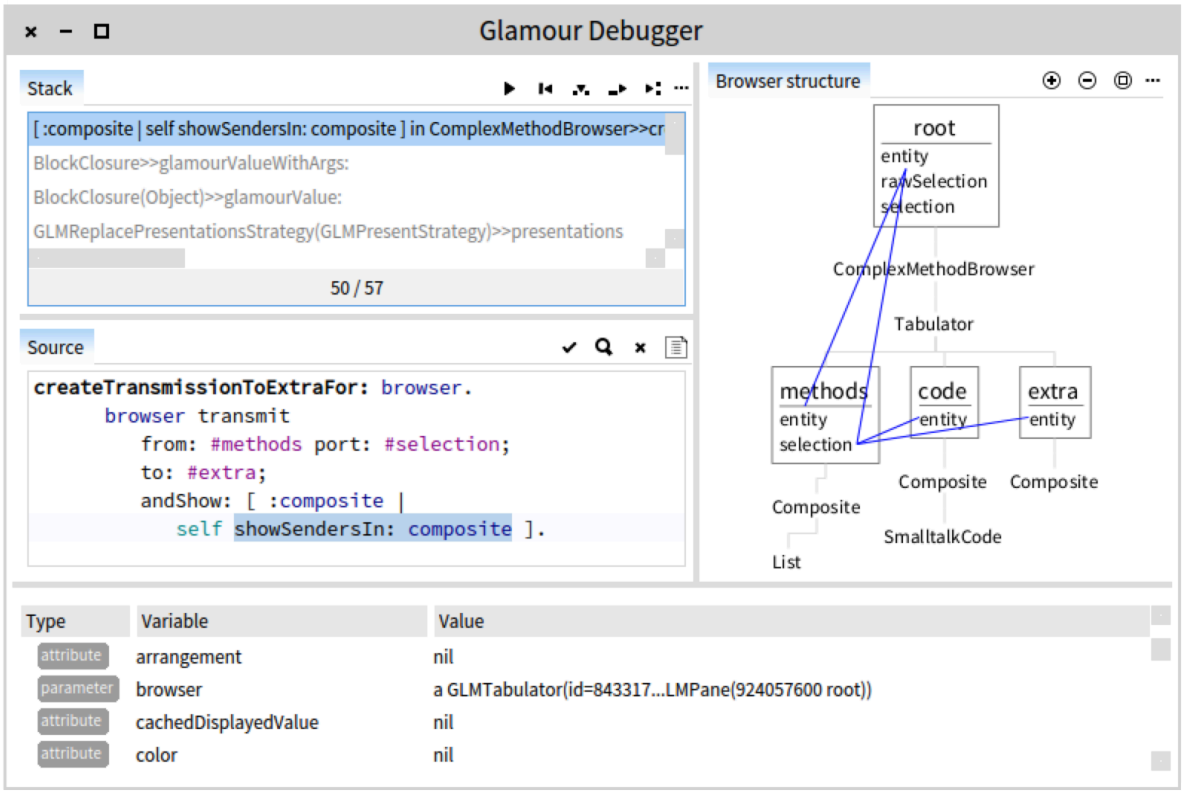

Figure 5: A domain-specific debugger for Glamour showing the model of the browser currently constructed.

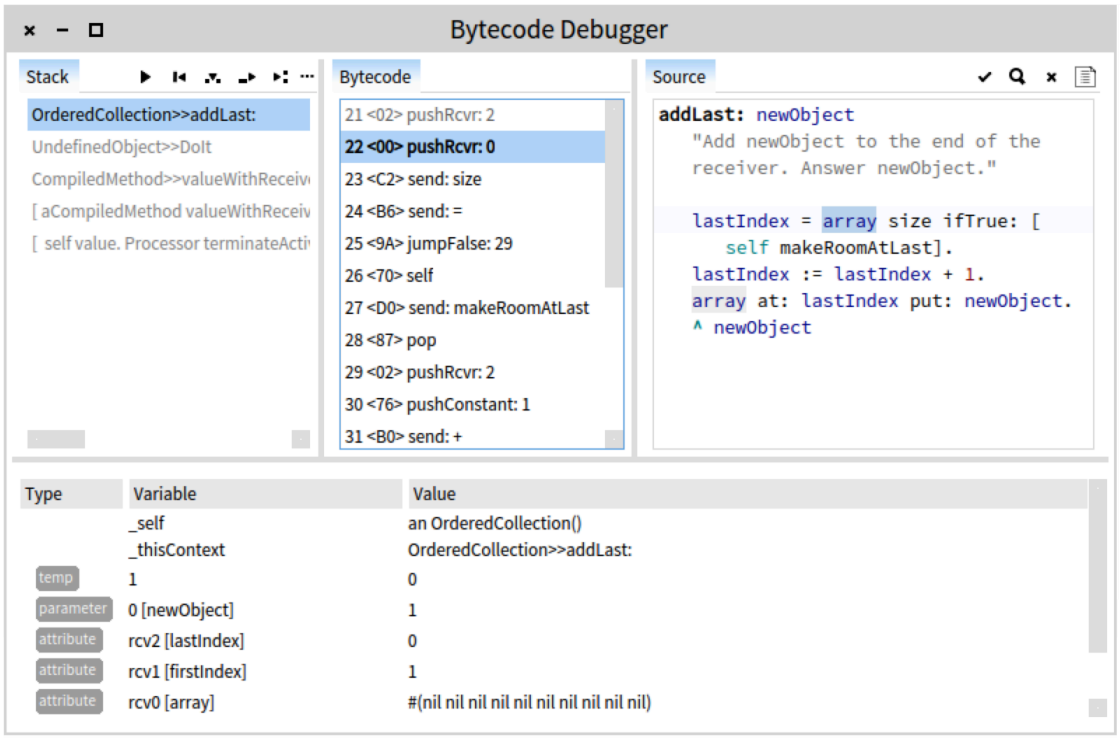

Figure 6: A domain-specific debugger for stepping through the execution of a program at the bytecode level.

4.6. Stepping through bytecodes

While normally programs are debugged at the source level, building tools like compilers and interpreters requires developers to understand the execution of a program at the bytecode level. However, all debugging actions presented in this section skip over multiple bytecode instructions. For example, *step into message send*, a debugging action present in most debuggers skips over the bytecode instructions that push method parameters onto the stack. The ability to debug at the bytecode level is especially important when generating or manipulating bytecode directly with bytecode transformation tools. In many cases the resulting bytecode cannot be de-compiled to a textual representation. To address this we developed a debugger for stepping through the execution of a program one bytecode instruction at a time:

Session: customizes the default debugging session to not step over multiple bytecode instructions when performing various initializations;

Predicates: no predicates are required

Operations:

Step over one bytecode instruction;

Step into a bytecode instruction representing a message send;

View: shows the method that is currently executed both as a list of bytecodes and a textual representation; embeds an object inspector that shows the internal stack of the current execution context (Figure 6);

Activation predicate: uses an activation predicate that always returns `true`.

4.7. Summary

PetitParser, Glamour, SUnit, Spy, Announcements framework and bytecode interpretation cover six distinct domains. For each one we instantiated a domain-specific debugger having a contextual debugging view and/or a set of debugging operations capturing abstractions from that domain. This shows the Moldable Debugger framework addresses the first two requirements.

The two remaining requirements, *automatic discovery* and *dynamic switching*, are also addressed. At each point during debugging developers can obtain a list of all domain-specific debuggers applicable to their current context. This does not require them either to know in advance all available debuggers, or to know when those debuggers are applicable. Once the right debugger was found developers can switch to it and continue debugging without having to restart the application. For example, one can perform the scenario presented in Section 2.4. The cost of creating these debuggers is discussed in Section 6.1.

5. Implementation aspects

While the Moldable Debugger model can capture a wide range of debugging problems the actual mechanism for detecting run-time events has a significant impact on the performance and usability of a debugger. In this section we present and discuss, from both a performance and usability point of view, three approaches for detecting run-time events in the execution of a debugged program based on debugging predicates:

(i) step-by-step execution;

(ii) code-centric instrumentation;

(iii) object-centric instrumentation.

5.1. Controlling the execution

When an event is detected a breakpoint is triggered, stopping the execution of the debugged program. The breakpoint notifies the active debugging action (*i.e.*, the action that installed the predicate). The debugging action can then perform an operation, resume the execution or wait for a user action.

We use the following debugging actions from the PetitParser debugger as examples in this section:

- *Production(aProduction)*: step until the given grammar production is reached;
- *Stream position(anInteger)*: step until parsing reaches a given position in the input stream.

5.1.1. Step-by-step execution

Approach. Interpret the debugged program one bytecode instruction at a time (*i.e.*, step-by-step execution) and check, after each bytecode instruction, if a debugging predicate matches the current execution context (*i.e.*, stack frame). This approach matches the *while-step* construct proposed by *Crawford et al.* [52].

Implementation. Each debugging predicate is transformed to a boolean condition that is applied to the current execution context (*i.e.*, stack frame).

Examples.

- *Production(aProduction)*: *(i)* check if the current bytecode instruction is the initial instruction of a method; *(ii)* check if the currently executing method is PPDelegateParser>>parseOn:; *(iii)* check if the receiver PPDelegateParser object is a parser for the given grammar production;
- *Stream position(anInteger)*: *(i)* check if the current bytecode instruction pushes a value into an object attribute; *(ii)* check if the attribute is named #stream and it belongs to an instance of PPStream; *(iii)* check if the value that will be pushed into the attribute is equal to the given value.

5.1.2. Code-centric instrumentation

Approach. Use basic debugging predicates (*i.e.*, *attribute access*, *attribute write*, *method call* and *message send*) to insert instrumentations into the code of the debugged program. State check predicates can then ensure that a breakpoint is triggered only if further conditions hold when the instrumentation is reached (*e.g.*, the target object is equal to a given one). This approach resembles dynamic aspects [53] and conditional breakpoints.

Implementation. We rely on two different mechanisms for handling predicates for attributes and methods.

We implement *attribute access* and *attribute write* predicates using slots [54]. Slots model instance variables as first-class objects that can generate the code for reading and writting instance variables. We rely on a custom slot that can wrap any existing slot and insert code for triggering a breakpoint before the attribute is read or written in all methods of a given class.

We implement *method call* and *message send* by adding meta-links to AST nodes [55]: when compiling an AST node to bytecode, if that AST node has an attached meta-link, that meta-link can generate code to be executed before, after or instead of the code represented by the AST node. We rely on a custom meta-link that inserts code for triggering a breakpoint before the execution of an AST node. We then implement these types of predicates as follows:

- *message send*: locate in the body of a method all AST nodes that represent a call to the target method; add the custom meta-link only to these AST nodes;
- *method call*: add the custom meta-link on the root AST node of the target method.

A different strategy for implementing code-centric instrumentations consists in injecting the debugged concern directly into an existing bytecode version of the code using a bytecode engineering library [56]. The meta-links used have similar properties: code instrumentation happens at runtime and the original code remains unchanged. Direct bytecode manipulation would give a more fined-grained control on the position where and how code inserted into the debugged code. This flexibility is not needed for our debugger and it would come with the cost of having to deal with the complexity of bytecode.

Examples.

- *Production(aProduction)*: instrument the root AST node of PPDelegateParser>>parseOn: to check if the receiver object is a parser for the given grammar production;
- *Stream position(anInteger)*: add instrumentations before all write instructions of the attribute #stream from class PPStream to check if the new value of the attribute is equal with a given value.

5.1.3. Object-centric instrumentation

Approach. Combine a basic predicate (*i.e.*, *attribute access*, *attribute write*, *method call* or *message send*) with a *state check* predicate verifying the identity of the receiver object against a given object (*i.e.*, *identity check* predicate). Then insert into the debugged program instrumentations only visible to the given object. Thus, a breakpoint is triggered only when the event captured by the basic predicate has taken place on the given instance. This approach matches the object-centric debugging approach [15], where debugging actions are specified at the object level (*e.g.*, stop execution when this object receives this message).

Implementation. We insert an instrumentation visible to only a single target object as follows:

- create an anonymous subclass of the target object's class; the anonymous subclass is created dynamically, at debug time, when the underlying dynamic action is executed;
- apply code-centric instrumentation to insert the basic debugging event into the anonymous class; code-centric instrumentation is inserted through code generation and recompilation of the anonymous class at debug time;

- change the class of the target object to the new anonymous class.

As access to a run-time object is necessary this approach can only be used once a debugger is present; it cannot be used to open the initial debugger.

Examples.

- *Production(aProduction)*: *(i)* locate the PPDelegateParser object that represents the given grammar production; *(ii)* replace the class of that object with an anonymous one where the method parseOn: has a *method call* predicate inserted using a code-centric instrumentation;

- *Stream position(anInteger)*: *(i)* locate the PPStream object holding the input that is being parsed; *(ii)* replace the class of that object with an anonymous one where the attribute #position: has an *attribute access* predicate inserted using a code-centric instrumentation.

5.2. Performance

To investigate the performance overhead of the aforementioned approaches we performed a series of micro-benchmarks. We performed these benchmarks using the current prototype of the Moldable Debugger[4,5] implemented in Pharo, an open-source Smalltalk environment, on an Apple MacBook Pro, 2.7 GHz Intel Core i7 in Pharo 4 with the jitted Pharo VM[6]. We ran each benchmark 5 times and present the average time of these runs in milliseconds. All presented times exclude garbage collection time.

5.2.1. Step-by-step execution

Basic benchmarks. We performed, for each basic predicate, a benchmark checking for an event (*i.e.*, method call/message send/attribute access/attribute/condition over the state) not present in the measured code. Given that a predicate never matches a point in the execution, the boolean condition will be checked for every bytecode instruction, giving us the worst-case overhead of these predicates on the measured code. The measured code(line 4)[7] consists of ten million calls to the method Counter>>#increment (lines 1-2). We selected this method as it has only one message send, attribute access and attribute write, making it possible to use it for all predicates (for *method call* we instrument the call to increment).

```
1 Counter>>#increment
2   counter := counter + 1
```

```
3 targetObject := Counter new.
4 10000000 timesRepeat: [targetObject increment]
```

As expected, this approach introduces a significant overhead of more than three orders of magnitude for all predicates, when the event of interest is not detected (Table 2). The high overhead is due to the step-by-step execution rather than to the actual condition being checked: verifying the identity of an object using a *state check* predicate (Table 2 – identity check) has the same overhead as a *state check* predicate that performs no check (Table 2 – empty state check).

Advanced benchmarks. To determine if this high overhead is present when dealing with real-world code, rather than a constructed example, we performed five more benchmarks presented in Listings 1 – 5. In each benchmark we used a *method call* predicate detecting the execution of a method not called in the measured code. Like in the previous benchmarks this gives us the maximal performance impact for this predicate. We only use one type of basic predicate given that all types of basic predicates exhibit a similar overhead.

[4] More details including demos and installation instructions can be found at: http://scg.unibe.ch/research/moldabledebugger

[5] A version of the prototype, including the code of the benchmarks can be found at http://scg.unibe.ch/download/moldabledebugger/prototype.zip

[6] http://files.pharo.org/vm/pharo

[7] In all code snippets showing code on which we performed a measurement, only the underlined line is the one that is actually being measured; the other lines represent just the setup and are not taken into account in the measurement.

Predicate	Normal execution	Step-by-step execution	Overhead
attribute access (#counter)	11 ms	13473 ms	1225×
attribute write (#counter)		13530 ms	1230×
method call (#increment)		14137 ms	1285×
message send (+)		14302 ms	1300×
identity check		12771 ms	1161×
empty state check		12627 ms	1148×

Table 2: Performance measurements done on simple examples for step-by-step execution.

Benchmark	Normal execution	Step-by-step execution	Overhead
factorial	1094 ms	2716 ms	2.6×
merge sort	4 ms	4530 ms	1120×
parser initialization	192 ms	5881 ms	37×
parser execution	18 ms	10334 ms	613×
announcement delivery	19 ms	16419 ms	864×

Table 3: Performance measurements of real-world examples for step-by-step execution.

Listing 1: Factorial

```
25000 factorial
```

Listing 2: Merge sort

```
collection := 2000 factorial asString.
collection sorted
```

Listing 3: Parser initialization: initialize a PetitParser parser for Java code

```
PPJavaParser new
```

Listing 4: Parser execution: parse the source code of the interface Stack[8] from Java using a parser for java code

```
parser := PPJavaParser new.
parserContext := PPContext new.
parserContext stream: self getStackJava.
parser parseWithContext: parserContext
```

Listing 5: Announcement delivery

```
announcer := Announcer new.
10000 timesRepeat: [
  announcer
    when: Announcement
    send: #execute:
    to: AnnouncementTarget new ].
announcer announce: Announcement
```

We obtained different results (Table 3) than in the previous set of benchmarks ranging from an overhead of only 2.6× to an overhead of 1120×. These diverging results can be explained by looking at one particular aspect of the measured code: the time spent in method calls that are implemented directly by the VM (*i.e.*, primitive operations) and thus cannot by executed in a step-by-step manner by a bytecode interpreter. For example, on the one hand, when computing factorial most time is spent doing additions, an operation implemented directly by the VM. Merge sort, on the other hand, spends little time in primitives; thus exhibits similar worst-case overhead to the example code from the previous benchmarks.

5.2.2. Code-centric instrumentation

This approach does not introduce any runtime overhead when using basic predicates to detect attribute reads/writes, message sends and method calls. The overhead comes from combining these predicates with state check predicates, and from the actual implementation mechanism used to check the condition. Given that we use two approaches for instrumenting code (*i.e.*, slots, AST annotations) we performed measurements that combine *attribute access* and *method call* predicates with *state check* predicates.

Basic benchmarks. We first combine the aforementioned predicates with an *identity check* predicate. For each situation we perform a benchmark on only the operation we are interested in (*i.e.*, attribute write – lines 20-21, method call to returnOne – lines 22-23) and on the #increment method used in the previous

[8] http://hg.openjdk.java.net/jdk6/jdk6/jdk/file/c228a234a3f3/src/share/classes/java/util/Stack.java

Instrumented method	Predicate	Normal execution	Instrumented execution	Overhead one check	Overhead three checks
#initializeWithOne	attribute write	81 ms	1317 ms	16×	17×
#returnOne	method call	83 ms	7664 ms	95×	98×
#increment	attribute write	103 ms	1350 ms	13×	14×
#increment	method call	103 ms	7560 ms	75×	77×
#initializeWithOne	attribute mutation	81 ms	645 ms	8×	-
#increment	attribute mutation	103 ms	652 ms	6×	-

Table 4: Performance measurements of simple examples for code-centric instrumentation.

section. We execute each method ten million times on one object and use an *identity check* predicate that never detects an event in the measured code (*i.e.*, checks for another object).

```
Counter>>#initializeWithOne
  counter := 1
```

```
Counter>>#returnOne
  ↑ 1
```

As seen from Table 4 the overhead is significantly lower than the one introduced by step-by-step execution. Regardless of the predicate, the highest overhead is obtained for the methods initializeWithOne and returnOne where, given that the methods have almost no functionality, any extra instrumentation increases execution time. The overhead for the increment method is lower as this method performs more operations than the previous two. Nevertheless, the *method call* predicate has an overhead six times higher than *attribute write* predicate. While for both implementations we reify the current stack frame before checking any associated condition, Reflectivity, the framework used for code instrumentation has an expensive mechanism for detecting recursive calls from meta-links (*i.e.*, detect when a meta-link is added in code called from the meta-link). Repeating these measurements when the basic predicates are combined with five identity check predicates results in only slightly higher overheads for all benchmarks. This indicates that most of the overhead comes from reifying the execution context every time a condition needs to be checked.

Based on the previous observation a further improvement can be done when combining an *attribute access/attribute write* predicate with a *state check* predicate that only accesses the new and old value of the instance variable: given that we use slots for instrumenting attributes accesses/writes we can directly get the new and old values of the attribute from the slot without reifying the current stack frame. This leads to a performance overhead just x8 when changes in the value of an attribute.

A further improvement in performance can be achieved by removing altogether the need for reifying the current stack frame. A *method call* predicate combined with an *identity check* predicate can directly insert in the code of the target method a condition comparing self (this in Java) with a given object. Our current prototype does not support these kinds of instrumentations. Nevertheless, the performance overhead required to reify the stack frame is small enough to have practical applicability. This approach further allows developers to test any kind of property of the stack frame (*e.g.*, the position of the program counter).

Advanced benchmarks. We performed four benchmarks on the following domain-specific actions presented in Section 4: *Production(aProduction)*, *Stream position(anInteger)*, *Parser(aParserClass)* and *Subscription(aSubscription)*. For the first three we used the code from Listing 4, while for the last one we used the code from Listing 5. For each debugging action we look for an event not present in the measured code (*e.g.*, a production not present in the parser).

For all debugging actions we get a runtime overhead lower than the one from the basic benchmarks ranging from 1.6x to 27.7x (Table 5). This is expected because in this case the event that triggers a breakpoint is encountered far less often. The debugging action *Parser(aParserClass)* has the largest overhead as it introduces a high number of instrumentations.

Debugging action	Normal execution	Instrumented execution	Overhead
Production(aProduction)	486	56	8.5×
Parser(aParser)	1553	56	27.7×
Stream position(anInteger)	92	56	1.65×
Subscription(aSubscription)	225	968	4.2×

Table 5: Performance measurements done on real-world examples for code-centric instrumentation.

5.2.3. Object-centric instrumentation

Using this approach there is no longer any runtime overhead in detecting when an attribute read/write, message send or method call happens on a given object. Runtime overhead is introduced by adding conditions that need to be checked when the target event is detected. For example, checking if a method is called on a target object that satisfies an extra condition only incurs runtime overhead for checking the extra condition every time the method is called on the target object.

Basic benchmarks. Given that we use code-centric instrumentations to insert those checks into the unique anonymous class of the target object, the performance overhead will always be lower than or equal to the overhead of just code-centric instrumentations. Consider the two situations below:

```
targetObject := Counter new.
10000000 timesRepeat: [
  targetObject initializeWithOne]
```

```
targetObjects := OrderedCollection new: 10000000.
10000000 timesRepeat: [
  targetObjects addLast: Counter new].
targetObjects do: [:aCounter| aCounter initializeWithOne]
```

In the code on the left, detecting when targetObject is initialized with value 2 has the same overhead as using code-centric instrumentations given that the condition must be checked on every write of the counter attribute (as seen in the previous section verifying the identity of one or more objects incurs a similar overhead, given that what takes the most time is reifying the execution context).

In the code on the right, the runtime overhead when checking that one object from the collection is initialized with 2 is negligible, as the condition is checked only once. Installing the predicate on every object will lead to a similar runtime overhead as in the previous case, given that the condition will be checked ten million times.

Advanced benchmarks. The same observations from *Basic benchmarks* apply. On the one hand, in the action *Stream position(anInteger)*, detecting when the stream has reached a certain position using an object-centric instrumentation has the same overhead as a code-centric instrumentation given that there is a single stream object shared by all the parsers. On the other hand, applying the action *Subscription(aSubscription)* on the code from Listing 5 has a negligible overhead as each announcement is delivered to a different object.

5.3. Usability

Step-by-step execution. The main advantage of this approach is that it is simple to understand and implement, and it does not alter the source of the debugged program. However, it can slow down the debugged program considerably to the point where it is no longer usable. Despite this shortcoming it can be useful for debugging actions that need to skip over a small number of bytecode instructions. For example, we use this approach to implement the action *Step to next subscription* in the Announcements debugger: we only need to skip over several hundred bytecode instructions internal to the Announcements framework.

Code-centric instrumentation. This approach has a much lower performance overhead than step-by-step execution that makes it practically applicable in most situations. While the current overhead is low, it can still prove inconvenient when using complex predicates or when stepping over code that already takes a significant amount of time. For example, we do not use this solution in the Announcements debugger as the Announcements framework is heavily used by Pharo IDE, and any overhead will apply to the entire IDE.

Object-centric instrumentation. While it imposes no performance overhead, this approach does not work for code that depends on the actual class of the instrumented object. It further requires access to an object beforehand, which is not always possible. We use this solution in the Announcements and PetitParser debuggers; however, in both cases we only instrument objects internal to these frameworks.

Discussion. Even if not practically applicable in most situations we used debugging actions based on step-by-step execution to implement the initial version of all our domain-specific debuggers. This allowed us to quickly prototype and refine the interface and the functionality of those debuggers. Later on, whenever performance became a problem we moved to actions based on code-centric instrumentation. We then only changed these actions to object-centric instrumentation in very specific situations where we could take advantage of particular aspects of a framework (*e.g.*, PetitParser uses a single PPContext object that is passed to all the parse methods; the Announcements framework creates an internal subscription object each time a subscriber registers with an announcer).

Depending on the particular aspects of a domain, not all three approaches are applicable. Table 6 indicates what approaches could be used for the example debuggers from Section 4 (Glamour is a prototype-based framework that relies on copying objects; Spy already instruments the debugged code).

	Step-by-step execution	Code-centric instrumentations	Object-centric instrumentations
Announcements	✓	✓	✓
Petit Parser	✓	✓	✓
Glamour	✓	✓	
Spy	✓		
Bytecode	✓		

Table 6: Feasible approaches for implementing debugging actions for the example debuggers from Section 4.

Note that the performance penalty is present only when using the custom debugger, and not when using the regular one.

5.4. The Moldable Debugger in other languages

The current prototype of the Moldable Debugger is implemented in Pharo. It can be ported to other languages as long as:

- they provide a debugging infrastructure that supports custom extensions/plugins for controlling the execution of a target program;
- there exists a way to rapidly construct user interfaces for debuggers, either through domain-specific languages or UI builders.

For example, one could implement the framework in Java. Domain-specific debugging operations can be implemented on top of the Java Debugging Interface (JDI) or by using aspects. JDI is a good candidate as it provides explicit control over the execution of a virtual machine and introspective access to its state. Aspect-Oriented Programming [57] can implement debugging actions by instrumenting only the code locations of interest. Dynamic aspects (*e.g.*, AspectWerkz [53]) can further scope code instrumentation at the debugger level. Last but not least, domain-specific views can be obtained by leveraging the functionality of IDEs, like *perspectives* in the Eclipse IDE.

6. Discussion

6.1. The cost of creating new debuggers

The four presented domain-specific debuggers were created starting from a model consisting of 1500 lines of code. Table 7 shows, for each debugger, how many lines of code were needed for the debugging view,

the debugging actions, and the debugging session. Regarding the view column, custom debuggers extend and customize the view of the default debugger; hence, the view of default debugger is twice the size of any other view, as it provides the common functionality needed in a debugging interface.

	Session	Operations	View	Total
Base model	800	700	-	1500
Default Debugger	-	100	400	500
Announcements	200	50	200	450
Petit Parser	100	300	200	600
Glamour	150	100	50	300
SUnit	100	-	50	150
Spy	-	30	30	60
Bytecode	20	50	130	200

Table 7: Size of extensions in lines of code (LOC).

In general, *lines of code* (LOC) must be considered with caution when measuring complexity and development effort. The metric does not necessarily indicate the time needed to write those lines. Nevertheless, it gives a good indication of the small size of these domain-specific debuggers. This small size makes the construction cost affordable. Similar conclusions can be derived from the work of *Kosar et al.* that shows that with the right setup it is possible to construct a domain-specific debugger for a modelling language with relatively low costs [58]. Hanson and Korn further show that a useful debugger for C can be written in under 2500 lines of code, one order of magnitude smaller than *gdb* [38].

The availability of a moldable infrastructure opens new possibilities:

(i) the developers of a library or framework can create and ship a dedicated debugger together with the code, to help users debug that framework or library. For example, the developers of PetitParser and Glamour can build custom debuggers themselves and ship them together with the frameworks;

(ii) developers can extend the debugger for their own applications, during the development process, to help them solve bugs or better understand the application. Smith *et al.* observed that developers take the initiative and build tools to solve problems they face during software development, even if they rarely share those tools [59].

6.2. Applicability

Section 4 shows that the Moldable Debugger can cover a wide range of application domains. While Section 4 just gives particular examples, we consider the Moldable Debugger to be applicable for most types of application domains that build upon an object-oriented model. For example, one could apply the proposed solution to AmbientTalk [60], an actor-based distributed programming language by extending the Moldable Debugger with support for actor-based concurrency, though it could require significant effort.

The applicability of the Moldable Debugger, nevertheless, has its limits. An edge case is Monaco, a domain-specific language for reactive systems with imperative programming notation [61]. While Monaco has a model based on hierarchical components that could be accommodated by the Moldable Debugger, the main goal of Monaco is to develop programs for control systems. As running and debugging programs on live control systems is not a feasible option, simulators, rather then debuggers provide better support for reasoning about theses types of program. A case where the Moldable Debugger would not be applicable is SymGridPar2, a language for parallel symbolic computation on a large number of cores [62]. On the one hand SymGridPar2 features a functional programming style. On the other hand it is designed for programs that will run in parallel on tens of thousand of cores. The run-time overhead added by a debugger can significantly influence the behaviour of the code. Logging frameworks provide better alternatives as they allow developers to collect information at run time with a very low overhead and analyze it postmortem with more costly analyses.

6.3. IDE Integration

Studies of software developers revealed that they use standalone tools alongside an IDE, even when their IDE has the required features, as they often cannot find those features [9]. Furthermore, developers also complain about loose integration of tools that forces them to look for relevant information in multiple places [63]. To avoid these problems the Moldable Debugger framework is integrated into the Pharo IDE and essentially replaces the existing debugger. On the one hand, Pharo made it easy to integrate the Moldable Debugger due to its powerful introspection support. For example, the entire run-time stack can be reified on demand and the class of an object can be changed dynamically at run time. Pharo further incorporates support for slots and behaviour reflection through AST annotations. On the other hand, due to the highly dynamic nature of Pharo/Smalltalk not all entry points for the debugger were clear, at the beginning.

The Moldable Debugger along with the domain-specific debuggers presented in Section 4 are also integrated into Moose[9], a platform for data and software analysis [64]. Despite the fact that the performance of the current implementation can be significantly improved, these domain-specific debuggers are usable and reduce debugging time. We are using the domain-specific debugger for PetitParser on a daily basis, as the default debugger significantly increases debugging time.

6.4. Pitfalls and limitations

Depending on the application domain and the actual debugger a developer wants to build, deeper knowledge about program execution may be needed. Hence, depending on the requirements, creating a domain-specific debugger is not an activity suitable for developers lacking this kind of knowledge.

Domain concepts, as well as the code that implements them can change. Given that we propose a manual approach for constructing domain-specific debuggers, these debuggers have to be manually kept in sync with both the domain concepts from an application and their actual implementation. Furthermore, bugs in the code of an application can introduce wrong behaviour in the debugger. For instance, a debugging predicate may not detect the desired run-time event due to remaining bugs in the code that needs to be debugged. Based on our experience in developing several domain-specific debuggers, a developer with domain knowledge can, with a low effort, detect these situations while developing a domain-specific debugger. The main issue consists in making sure that new changes in an application do not break assumptions made when developing a debugger. Currently, together with each debugger we also created smoke tests that exercise the main functionality of that debugger. This gives a minimum safety-net for detecting changes that break the debugger. A wider range of tests that exercises different usage scenarios could detect significantly more problematic changes. While writing these tests by hand, for each debugger, could be an option, we also see automatic test generation based on more meta-information as a possible solution, though we currently did not investigate this alternative.

6.5. Open questions

As software systems evolve domain-specific debuggers written for those systems must also evolve. This raises further research questions like: *"What changes in the application will lead to changes in the debugger?"* or *"How can the debugger be kept in sync with the application?"*. For example, introducing code instrumentation or destructive data reading (as in a stream) can lead to significant changes in an existing debugger.

In this context, a more high-level question is *"What makes an application debuggable?"*. By this we mean what characteristics of an application ease, or exacerbate the creation of debuggers or, more generally, what characteristics affect debugging. To draw an analogy, in the field of software testing high coupling makes the creation of unit tests difficult (by increasing the number of dependencies that need to be taken into account) and thus decreases the testability of a software system.

[9] http://moosetechnology.org

7. Related Work

There exists a wide body of research addressing debugging from various perspectives. In this section we give an overview of several perspectives related to the Moldable Debugger model.

7.1. Software logging

While debuggers aim to support a direct interaction with the run-time state of an application, logging frameworks just record dynamic information about the execution of a program. One important challenge with logging frameworks, related to the current work, is how to capture the right information about the run-time [65]. Given the large diversity of information that could be relevant about a running application, like the Moldable Debugger, many frameworks for logging run-time information allow developer to customize the logging infrastructure to fit their needs [7, 66, 8]. MetaSpy, for example, makes is possible to easily create domain specific-profilers [8].

In the context of wireless sensor networks Cao *et al.* propose declarative tracepoints, a debugging system that allows users to insert action-associated checkpoints using a SQL-like language. Like activation predicates it allows users to detect and log run-time events using *condition predicates* over the state of the program [67].

7.2. Specifying domain-specific operations

There is a wide body of research in programmable/scriptable debugging allowing developers to automate debugging tasks by creating high-level abstractions. *MzTake* [26] is a scriptable debugger enabling developers to automate debugging tasks, inspired by *functional reactive programming*. *MzTake* treats a running program as a stream of run-time events that can be analyzed using operators, like *map* and *filter*; streams can be combined to form new streams. For example, one can create a stream in which a new value is added every time a selected method is called from the debugged program. Selecting only method calls performed on objects that are in a certain state is achieved using the *filter* operator; this operator creates a new stream that contains only the method call events that matched the filter's condition. Unlike *MzTake* we propose an approach for detecting run-time events based on object-oriented constructs: run-time events are specified by combining predicate objects, instead of combining streams. A debugging action can use then a predicate to detect a run-time event (*e.g.*, method call on a given object) and put the event in a stream.

Coca [68] is an automated debugger for C using Prolog predicates to search for events of interest over program state. Events capture various language constructs (*e.g.*, function, return, break, continue, goto) and are modelled as C structures; a sequence of events is grouped in a trace. Developers write, using Prolog, queries that search for an event matching a pattern in a trace. To perform a query a developer has to provide an *event pattern* and call a primitive operation for performing the actual search. The event patterns consists of any combination of 3-tuples of the form '<*attributename*> <*operation*> <*attribute-value*>' connected with *and*, *or* or *not*. In our approach we express high-level run-time events by combining objects (*i.e.*, debugging predicates) instead of combining tuples. We further use predicates as a specification of run-time events and employ different implementations to detect those events at run-time.

Dalek [25] is a C debugger employing a dataflow approach for debugging sequential programs: developers create high-level events by combining primitive events and other high-level events. Developers enter breakpoints into the debugged code to generate primitive events. A high-level event is created by specifying in the definition of that event what events activate that event; when an event is triggered all other events that depend on it are also triggered. High-level events can maintain state and execute code when triggered (*e.g.*, print, test invariants). Thus, high-level events map to debugging actions in our approach. However, we do not require developers to explicitly trigger primitive events from the debugged code; developers provide a specification of the run-time event using debugging predicates, outside of the debugged program's code.

Auguston *et al.* present a framework that uses declarative specifications of debugging actions over event traces to monitor the execution of a program [33]. Several types of run-time events, corresponding to various actions encountered in the debugged program, are generated directly by the virtual machine. Events are grouped together in traces that conform to an event grammar, defining the valid traces of events. An execution monitor loads a target program, executes it, obtains a trace of run-time events from the program

and performs various computations over the event trace (*e.g.*, check invariants, profile). We do not have an explicit concept of monitor in our approach and do not directly provide operations for manipulating event traces. Our model only associates run-time events (predicates) with various operations. Event traces can be implemented on top of this model, by having debugging actions that store and manipulate events.

Expositor [27] is a scriptable time-travel debugger that can check temporal properties of an execution: it views program traces as immutable lists of time-annotated program state snapshots and uses an efficient data structure to manage them. *Acid* [69] makes it possible to write debugging operations, like breakpoints and step instructions, in a language designed for debugging that reifies program state as variables.

The aforementioned approaches focus on improving debugging by enabling developers to create commands, breakpoints or queries at a higher level of abstraction. Nevertheless, while they encapsulate high-level abstractions into scripts, programs or files, developers have to manually find proper high-level abstractions for a given debugging context. We propose an approach for automatically detecting relevant high-level abstractions (*e.g.*, debugging actions) based on the current debugging context. Furthermore, only some of the aforementioned approaches incorporate at least ad hoc possibilities of visually exploring data by using features from the host platform. We propose a domain-specific view for each domain-specific debugger that displays and groups relevant widgets for the current debugging context.

Object-centric debugging [15] proposes a new way to perform debugging operations by focusing on objects instead of the execution stack; while it increases the level of abstraction of debugging actions to object-oriented idioms, the approach does not enable developers to create and combine debugging actions to capture domain concepts instead of just object-oriented idioms. *Reverse watchpoints* use the concept of *position* to automatically find the last time a target variable was written and move control flow to that point [70]. *Whyline* is a debugging tool that allows developer to ask and answer *Why* and *Why Not* questions about program behavior [71]. *Query-based debugging* facilitates the creation of queries over program execution and state using high-level languages [28, 29, 30, 31]. *Duel* [72] is a high-level language on top of GDB for writing state exploration queries. These approaches are complementary to our approach as they can be used to create other types of debugging operations.

Omniscient debugging provides a way to navigate backwards in time within a program's execution trace [3]. Bousse *et al.* introduce an omniscient debugger targeting an executable Domain-Specific Modeling Language (xDSML) [73], that while still partially generic can generate domain-specific traces tuned to the actual xDSML. This debugger is partially generic as it treats all xDSMLs in a uniform way. Like our approach this debugger is based on an object-oriented model and aims to improve the debugging process by presenting the user with domain-specific information. The Moldable Debugger, however, is not omniscient but allows developers to fine tune the debugger to the particular aspects of a domain-specific language or application.

Lee *et al.* propose a debugger model for composing portable mixed-environment debuggers [74]. Their current implementation, *Blink*, is a full-featured debugger for both Java and C. While the Moldable Debugger model does not depend on a particular object-oriented language, we do not provide an approach for cross-language debugging (*e.g.*, between an object-oriented and a non object-oriented language).

7.3. User interfaces for debugging

Another category of approaches looks at how to improve the user interface of debuggers instead of their actions. *Debugger Canvas* [32] proposes a novel type of user interface for debuggers based on the *Code Bubbles* [75] paradigm. Rather than starting from a user interface having a predefined structure, developers start from an empty one on which different bubbles are added, as they step through the execution of the program. Our approach requires developers to create custom user interfaces (views) beforehand. *The Data Display Debugger (DDD)* [76] is a graphical user interface for GDB providing a graphical display for representing complex data structures as graphs that can be explored incrementally and interactively. *jGRASP* supports the visualization of various data structure by means of dynamic viewers and a *structure identifier* that automatically select suitable views for data structures [77]. *x*DIVA is a 3-D debugging visualization system where complex visualization metaphors are assembled from individual ones, each of which is independently replaceable [78].

Each of these approaches introduces different improvements in the user interface of a debugger. To take advantage of this our approach does not hardcode the user interface of the debugger: each domain-specific debugger can have a dedicated user interface. Given that domain-specific debuggers are switchable at run time, when multiple debuggers are applicable a developer can select the one whose user interface she finds appropriate. By focusing only on the user interface these approaches do not provide support for adding custom debugging operations. Our approach addresses both aspects.

7.4. Unifying approaches

deet [38] is a debugger for ANSI C written in *tksh*[10] that, like our approach, promotes simple debuggers having few lines of code, and allows developers to extend the user interface and add new commands by writing code in a high-level language (*i.e.*, *tksh*). Commands are directly embedded in the user interface. Our approach decouples debugging actions from user-interface components (*i.e.*, widgets): each widget dynamically loads at run time debugging actions that have a predefined annotation. If in *deet* run-time events are detected by attaching *tksh* code to conditional breakpoints, we provide a finer model based on combining debugging predicates. Last but not least, we propose modelling the customization of a debugger (*i.e.*, debugging actions + user interface) through explicit domain-specific extensions and provide support for automatically detecting appropriate extensions at run time. In *deet* developers have to manually select appropriate debuggers.

TIDE is a debugging framework focusing on the instantiation of debuggers for formal languages (ASF+SDF, in particular) [79]; developers can implement high-level debugging actions like breakpoints and watchpoints, extend the user interface by modifying the Java implementation of TIDE, and use *debugging rules* to state which debugging actions are available at which logical breakpoints. *LISA* is a grammar-based compiler generator that can automatically generate debuggers, inspectors and visualizers for DSLs that have a formal language specification [80]. Debuggers are obtained by constructing, from the grammar, transformations mapping from the DSL code to the generated GPL code. Wu *et al.* present a grammar-driven technique for automatically generating debuggers for DSLs implemented using source-to-source translation (a line of code from a DSL is translated into multiple consecutive lines of GPL code); this makes it possible to reuse an existing GPL debugger [18]. Other language workbenches [81, 19, 82] for developing DSLs or language extensions follow similar ideas: they enable the creator of a DSL or language extension to provide extra specifications during the development of the DSL or language extension that are then used to generate a specialized debugger. Our approach targets object-oriented applications where a formal specification is missing and not programs written in domain-specific languages that have a grammar or another formal specification. Furthermore, if domain concepts are built on top of a DSL, then DSL debuggers suffer from the same limitations as generic debuggers. Our approach directly supports debuggers aware of application domains.

7.5. Debugging in domain-specific modelling

Domain-specific modelling (DSM) enables domain-experts to directly work with familiar concepts instead of manually mapping concepts from the problem domain to the solution domain. Debuggers that work at a lower level of abstraction than that of the model limit the ability of a domain-expert to properly debug a model. To address this, the goal of domain-specific modelling is to automatically generate debuggers from a meta-model. While most meta-modelling tools do not automatically generate debuggers, several approaches approach this goal. Mannadiar and Vangheluwe propose a conceptual mapping between debugging concepts from the programming languages (*e.g.*, breakpoints, assertions) and concepts from domain-specific modelling languages that use rule-based approaches for model transformations [83]. Kosar *et al.* discuss debugging facilities for a modelling environment for measurement systems [58]. Kolomvatsos *et al.* present a debugger architecture for a domain-specific language used to model autonomous mobile nodes [20]. These approaches take advantage of, and integrate with meta-modeling tools. The approach proposed in this paper is for object-oriented applications where the model consists of objects and relations between them. This model is

[10] An extension of Korn shell including the graphical support for Tcl/Tk.

created by application developers using idioms provided directly by object-oriented programming without the use of a meta-modelling tool.

8. Conclusions

Developers encounter domain-specific questions. Traditional debuggers relying on generic mechanisms, while universally applicable, are less suitable to handle domain-specific questions. The Moldable Debugger addresses this contradiction by allowing developers to create with little effort domain-specific debuggers that enable custom debugging actions through custom user interfaces. As a validation, we implemented the Moldable Debugger model and created six different debuggers in fewer than 600 lines of code each. Through these examples we show that the Moldable Debugger can reduce the abstraction gap between the debugging needs and debugging support leading to a more efficient and less error-prone debugging effort.

We further explored three approaches for implementing debugging actions. Performance can be impacted, but only when in the special debugging mode. For the regular debugging mode there is no penalty. Thus, our solution offers an opt-in possibility with no cost.

Given the large costs associated with debugging activities, improving the workflow and reducing the cognitive load of debugging can have a significant practical impact. With our approach developers can create their own debuggers to address recurring problems. These custom debuggers come with a price as they have to be constructed by application or framework developers rather than by tool providers. Nevertheless, this can make considerable economical sense when working on a long-lived system. Furthermore, library developers can ship library-specific debuggers together with their product. This can have a practical impact due to the reuse of the library in many applications.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project Nr. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015). We thank Alexandre Bergel, Jorge Ressia, Mircea Lungu, and the anonymous reviewers for their suggestions in improving this paper.

References

[1] I. Vessey, Expertise in debugging computer programs: An analysis of the content of verbal protocols, IEEE Trans. Syst. Man Cybern. 16 (5) (1986) 621–637. doi:10.1109/TSMC.1986.289308.
URL http://dx.doi.org/10.1109/TSMC.1986.289308

[2] G. Tassey, The economic impacts of inadequate infrastructure for software testing, Tech. rep., National Institute of Standards and Technology (2002).

[3] G. Pothier, E. Tanter, J. Piquer, Scalable omniscient debugging, Proceedings of the 22nd Annual SCM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07) 42 (10) (2007) 535–552. doi:10.1145/1297105.1297067.

[4] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, S. Pasupathy, Sherlog: Error diagnosis by connecting clues from run-time logs, SIGARCH Comput. Archit. News 38 (1) (2010) 143–154. doi:10.1145/1735970.1736038.
URL http://doi.acm.org/10.1145/1735970.1736038

[5] S. Han, Y. Dang, S. Ge, D. Zhang, T. Xie, Performance debugging in the large via mining millions of stack traces, in: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 145–155.
URL http://dl.acm.org/citation.cfm?id=2337223.2337241

[6] A. Zeller, Yesterday, my program worked. Today, it does not. Why?, in: ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering, Springer-Verlag, London, UK, 1999, pp. 253–267. doi:10.1145/318773.318946.

[7] Log4j (accessed June 11, 2015).
URL http://logging.apache.org/log4j

[8] J. Ressia, A. Bergel, O. Nierstrasz, L. Renggli, Modeling domain-specific profilers, Journal of Object Technology 11 (1) (2012) 1–21. doi:10.5381/jot.2012.11.1.a5.
URL http://www.jot.fm/issues/issue_2012_04/article5.pdf

[9] T. Roehm, R. Tiarks, R. Koschke, W. Maalej, How do professional developers comprehend software?, in: Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012, IEEE Press, Piscataway, NJ, USA, 2012, pp. 255–265.

[10] G. C. Murphy, M. Kersten, L. Findlater, How are Java software developers using the Eclipse IDE?, IEEE Softw. 23 (4) (2006) 76–83. doi:10.1109/MS.2006.105.
URL http://dx.doi.org/10.1109/MS.2006.105

[11] K. Beck, Test Driven Development: By Example, Addison-Wesley Longman, 2002.

[12] D. C. Littman, J. Pinto, S. Letovsky, E. Soloway, Mental models and software maintenance, J. Syst. Softw. 7 (4) (1987) 341–355. doi:10.1016/0164-1212(87)90033-1.
URL http://dx.doi.org/10.1016/0164-1212(87)90033-1

[13] V. Rajlich, N. Wilde, The role of concepts in program comprehension, in: Proceedings of the 10th International Workshop on Program Comprehension, IWPC '02, IEEE Computer Society, Washington, DC, USA, 2002, pp. 271–.
URL http://dl.acm.org/citation.cfm?id=580131.857012

[14] A. Zeller, Why Programs Fail: A Guide to Systematic Debugging, Morgan Kaufmann, 2005.

[15] J. Ressia, A. Bergel, O. Nierstrasz, Object-centric debugging, in: Proceedings of the 34rd international conference on Software engineering, ICSE '12, 2012. doi:10.1109/ICSE.2012.6227167.
URL http://scg.unibe.ch/archive/papers/Ress12a-ObjectCentricDebugging.pdf

[16] A. Lienhard, T. Gîrba, O. Nierstrasz, Practical object-oriented back-in-time debugging, in: Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08), Vol. 5142 of LNCS, Springer, 2008, pp. 592–615, ECOOP distinguished paper award. doi:10.1007/978-3-540-70592-5_25.
URL http://scg.unibe.ch/archive/papers/Lien08bBackInTimeDebugging.pdf

[17] J. Sillito, G. C. Murphy, K. De Volder, Asking and answering questions during a programming change task, IEEE Trans. Softw. Eng. 34 (2008) 434–451. doi:10.1109/TSE.2008.26.
URL http://portal.acm.org/citation.cfm?id=1446226.1446241

[18] H. Wu, J. Gray, M. Mernik, Grammar-driven generation of domain-specific language debuggers, Softw. Pract. Exper. 38 (10) (2008) 1073–1103. doi:10.1002/spe.v38:10.
URL http://dx.doi.org/10.1002/spe.v38:10

[19] R. T. Lindeman, L. C. Kats, E. Visser, Declaratively defining domain-specific language debuggers, SIGPLAN Not. 47 (3) (2011) 127–136. doi:10.1145/2189751.2047885.
URL http://doi.acm.org/10.1145/2189751.2047885

[20] K. Kolomvatsos, G. Valkanas, S. Hadjiefthymiades, Debugging applications created by a domain specific language: The IPAC case, J. Syst. Softw. 85 (4) (2012) 932–943. doi:10.1016/j.jss.2011.11.1009.
URL http://dx.doi.org/10.1016/j.jss.2011.11.1009

[21] A. Blunk, J. Fischer, D. A. Sadilek, Modelling a debugger for an imperative voice control language, in: Proceedings of the 14th International SDL Conference on Design for Motes and Mobiles, SDL'09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 149–164.
URL http://dl.acm.org/citation.cfm?id=1812885.1812899

[22] J. Rumbaugh, I. Jacobson, G. Booch, Unified Modeling Language Reference Manual, The (2nd Edition), Pearson Higher Education, 2004.

[23] M. Fowler, Domain-Specific Languages, Addison-Wesley Professional, 2010.

[24] L. Renggli, T. Gîrba, O. Nierstrasz, Embedding languages without breaking tools, in: T. D'Hondt (Ed.), ECOOP'10: Proceedings of the 24th European Conference on Object-Oriented Programming, Vol. 6183 of LNCS, Springer-Verlag, Maribor, Slovenia, 2010, pp. 380–404. doi:10.1007/978-3-642-14107-2_19.
URL http://scg.unibe.ch/archive/papers/Reng10aEmbeddingLanguages.pdf

[25] R. A. Olsson, R. H. Crawford, W. W. Ho, A dataflow approach to event-based debugging, Softw. Pract. Exper. 21 (2) (1991) 209–229. doi:10.1002/spe.4380210207.
URL http://dx.doi.org/10.1002/spe.4380210207

[26] G. Marceau, G. H. Cooper, J. P. Spiro, S. Krishnamurthi, S. P. Reiss, The design and implementation of a dataflow language for scriptable debugging, Automated Software Engg. 14 (1) (2007) 59–86. doi:10.1007/s10515-006-0003-z.
URL http://dx.doi.org/10.1007/s10515-006-0003-z

[27] Y. P. Khoo, J. S. Foster, M. Hicks, Expositor: scriptable time-travel debugging with first-class traces, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 352–361.
URL http://dl.acm.org/citation.cfm?id=2486788.2486835

[28] R. Lencevicius, U. Hölzle, A. K. Singh, Query-based debugging of object-oriented programs, in: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming (OOPSLA'97), ACM, New York, NY, USA, 1997, pp. 304–317. doi:10.1145/263698.263752.

[29] A. Potanin, J. Noble, R. Biddle, Snapshot query-based debugging, in: Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04), IEEE Computer Society, Washington, DC, USA, 2004, p. 251.

[30] M. Martin, B. Livshits, M. S. Lam, Finding application errors and security flaws using PQL: a program query language, in: Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05), ACM Press, New York, NY, USA, 2005, pp. 363–385.

[31] S. Ducasse, T. Gîrba, R. Wuyts, Object-oriented legacy system trace-based logic testing, in: Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06), IEEE Computer Society Press, 2006, pp. 35–44. doi:10.1109/CSMR.2006.37.

URL http://scg.unibe.ch/archive/papers/Duca06aTestLogtestingCSMR.pdf
[32] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, S. P. Reiss, Debugger canvas: industrial experience with the code bubbles paradigm, in: Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012, IEEE Press, Piscataway, NJ, USA, 2012, pp. 1064–1073.
URL http://dl.acm.org/citation.cfm?id=2337223.2337362
[33] M. Auguston, C. Jeffery, S. Underwood, A framework for automatic debugging., in: ASE, IEEE Computer Society, 2002, pp. 217–222.
[34] Pharo Programming Language (accessed June 11, 2015).
URL http://pharo.org
[35] A. Chiş, T. Gîrba, O. Nierstrasz, The Moldable Debugger: A framework for developing domain-specific debuggers, in: B. Combemale, D. Pearce, O. Barais, J. J. Vinju (Eds.), Software Language Engineering, Vol. 8706 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 102–121. doi:10.1007/978-3-319-11245-9_6.
URL http://scg.unibe.ch/archive/papers/Chis14b-MoldableDebugger.pdf
[36] A. Ko, B. Myers, M. Coblenz, H. Aung, An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks, Software Engineering, IEEE Transactions on 32 (12) (2006) 971 –987. doi:10.1109/TSE.2006.116.
[37] M. Kersten, G. C. Murphy, Mylar: a degree-of-interest model for ides, in: AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, ACM Press, New York, NY, USA, 2005, pp. 159–168. doi:10.1145/1052898.1052912.
[38] D. R. Hanson, J. L. Korn, A simple and extensible graphical debugger, in: IN WINTER 1997 USENIX CONFERENCE, 1997, pp. 173–184.
[39] E. Murphy-Hill, C. Parnin, A. P. Black, How we refactor, and how we know it, in: Proceedings of the 31st International Conference on Software Engineering, ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 287–297. doi:10.1109/ICSE.2009.5070529.
[40] E. Murphy-Hill, R. Jiresal, G. C. Murphy, Improving software developers' fluency by recommending development environment commands, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, ACM, New York, NY, USA, 2012, pp. 42:1–42:11. doi:10.1145/2393596.2393645.
URL http://doi.acm.org/10.1145/2393596.2393645
[41] W.-P. d. Roever, K. Engelhardt, Data Refinement: Model-Oriented Proof Methods and Their Comparison, 1st Edition, Cambridge University Press, New York, NY, USA, 2008.
[42] K. Beck, Kent Beck's Guide to Better Smalltalk, Sigs Books, 1999.
[43] A. Chiş, O. Nierstrasz, T. Gîrba, Towards a moldable debugger, in: Proceedings of the 7th Workshop on Dynamic Languages and Applications, 2013. doi:10.1145/2489798.2489801.
URL http://scg.unibe.ch/archive/papers/Chis13a-TowardsMoldableDebugger.pdf
[44] L. Renggli, S. Ducasse, T. Gîrba, O. Nierstrasz, Practical dynamic grammars for dynamic languages, in: 4th Workshop on Dynamic Languages and Applications (DYLA 2010), Malaga, Spain, 2010, pp. 1–4.
URL http://scg.unibe.ch/archive/papers/Reng10cDynamicGrammars.pdf
[45] ANTLR – ANother Tool for Language Recognition (accessed June 11, 2015).
URL http://www.antlr.org/
[46] ANTLR – Debugging ANTLR grammars using ANTLR Studio (accessed June 11, 2015).
URL http://www.placidsystems.com/articles/article-debugging/usingdebugger.htm
[47] D. Rebernak, M. Mernik, H. Wu, J. G. Gray, Domain-specific aspect languages for modularising crosscutting concerns in grammars, IET Software 3 (3) (2009) 184–200. doi:10.1049/iet-sen.2007.0114.
URL http://dx.doi.org/10.1049/iet-sen.2007.0114
[48] P. Bunge, Scripting browsers with Glamour, Master's thesis, University of Bern (Apr. 2009).
URL http://scg.unibe.ch/archive/masters/Bung09a.pdf
[49] A. Bergel, F. Bañados, R. Robbes, D. Röthlisberger, Spy: A flexible code profiling framework, Journal of Computer Languages, Systems and Structures 38 (1) (2012) 16 – 28, {SMALLTALKS} 2010. doi:10.1016/j.cl.2011.10.002.
URL http://www.sciencedirect.com/science/article/pii/S1477842411000327
[50] J. Brant, B. Foote, R. Johnson, D. Roberts, Wrappers to the rescue, in: Proceedings European Conference on Object Oriented Programming (ECOOP'98), Vol. 1445 of LNCS, Springer-Verlag, 1998, pp. 396–417.
[51] S2py Profiling Framework (accessed June 11, 2015).
URL http://www.smalltalkhub.com/#!/~ObjectProfile/S2py
[52] R. H. Crawford, R. A. Olsson, W. W. Ho, C. E. Wee, Semantic issues in the design of languages for debugging, Comput. Lang. 21 (1) (1995) 17–37. doi:10.1016/0096-0551(94)00015-I.
URL http://dx.doi.org/10.1016/0096-0551(94)00015-I
[53] J. Bonér, What are the key issues for commercial AOP use: how does AspectWerkz address them?, in: Proceedings of the 3rd international conference on Aspect-oriented software development, AOSD '04, ACM, New York, NY, USA, 2004, pp. 5–6. doi:10.1145/976270.976273.
[54] T. Verwaest, C. Bruni, M. Lungu, O. Nierstrasz, Flexible object layouts: enabling lightweight language extensions by intercepting slot access, in: Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications, OOPSLA '11, ACM, New York, NY, USA, 2011, pp. 959–972. doi:10.1145/2048066.2048138.
URL http://scg.unibe.ch/archive/papers/Verw11bFlexibleObjectLayouts.pdf
[55] M. Denker, S. Ducasse, A. Lienhard, P. Marschall, Sub-method reflection, in: Journal of Object Technology, Special Issue.

Proceedings of TOOLS Europe 2007, Vol. 6/9, ETH, 2007, pp. 231–251. doi:10.5381/jot.2007.6.9.a14.
URL http://www.jot.fm/contents/issue_2007_10/paper14.html
[56] É. Tanter, M. Ségura-Devillechaise, J. Noyé, J. Piquer, Altering Java semantics via bytecode manipulation, in: Proceedings of GPCE'02, Vol. 2487 of LNCS, Springer-Verlag, 2002, pp. 283–89.
[57] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: M. Aksit, S. Matsuoka (Eds.), ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming, Vol. 1241 of LNCS, Springer-Verlag, Jyvaskyla, Finland, 1997, pp. 220–242. doi:10.1007/BFb0053381.
[58] T. Kosar, M. Mernik, J. Gray, T. Kos, Debugging measurement systems using a domain-specific modeling language, Computers in Industry 65 (4) (2014) 622 – 635. doi:10.1016/j.compind.2014.01.013.
URL http://www.sciencedirect.com/science/article/pii/S0166361514000293
[59] E. K. Smith, C. Bird, T. Zimmermann, Build it yourself! homegrown tools in a large software company, in: Proceedings of the 37th International Conference on Software Engineering, IEEE – Institute of Electrical and Electronics Engineers, 2015.
URL http://research.microsoft.com/apps/pubs/default.aspx?id=238936
[60] T. V. Cutsem, E. G. Boix, C. Scholliers, A. L. Carreton, D. Harnie, K. Pinte, W. D. Meuter, AmbientTalk: programming responsive mobile peer-to-peer applications with actors, Computer Languages, Systems and Structures 40 (3–4) (2014) 112–136. doi:10.1016/j.cl.2014.05.002.
URL http://www.sciencedirect.com/science/article/pii/S1477842414000335
[61] H. Prähofer, R. Schatz, C. Wirth, D. Hurnaus, H. Mössenböck, Monaco–a domain-specific language solution for reactive process control programming with hierarchical components, Computer Languages, Systems and Structures 39 (3) (2013) 67–94. doi:10.1016/j.cl.2013.02.001.
URL http://www.sciencedirect.com/science/article/pii/S1477842413000031
[62] P. Maier, R. Stewart, P. Trinder, Reliable scalable symbolic computation: The design of SymGridPar2, Computer Languages, Systems and Structures 40 (1) (2014) 19–35, special issue on the Programming Languages track at the 28th {ACM} Symposium on Applied Computing. doi:10.1016/j.cl.2014.03.001.
URL http://www.sciencedirect.com/science/article/pii/S1477842414000049
[63] W. Maalej, Task-first or context-first? Tool integration revisited, in: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 344–355. doi:10.1109/ASE.2009.36.
URL http://dx.doi.org/10.1109/ASE.2009.36
[64] O. Nierstrasz, S. Ducasse, T. Gîrba, The story of Moose: an agile reengineering environment, in: Proceedings of the European Software Engineering Conference (ESEC/FSE'05), ACM Press, New York, NY, USA, 2005, pp. 1–10, invited paper. doi:10.1145/1095430.1081707.
URL http://scg.unibe.ch/archive/papers/Nier05cStoryOfMoose.pdf
[65] A. Oliner, A. Ganapathi, W. Xu, Advances and challenges in log analysis, Commun. ACM 55 (2) (2012) 55–61. doi:10.1145/2076450.2076466.
URL http://doi.acm.org/10.1145/2076450.2076466
[66] U. Erlingsson, M. Peinado, S. Peter, M. Budiu, Fay: Extensible distributed tracing from kernels to clusters, in: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, ACM, New York, NY, USA, 2011, pp. 311–326. doi:10.1145/2043556.2043585.
URL http://doi.acm.org/10.1145/2043556.2043585
[67] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, L. Luo, Declarative tracepoints: A programmable and application independent debugging system for wireless sensor networks, in: Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems, SenSys '08, ACM, New York, NY, USA, 2008, pp. 85–98. doi:10.1145/1460412.1460422.
URL http://doi.acm.org/10.1145/1460412.1460422
[68] M. Ducassé, Coca: An automated debugger for C, in: International Conference on Software Engineering, 1999, pp. 154–168.
[69] P. Winterbottom, ACID: A debugger built from a language, in: USENIX Technical Conference, 1994, pp. 211–222.
[70] K. Maruyama, M. Terada, Debugging with reverse watchpoint, in: Proceedings of the Third International Conference on Quality Software (QSIC'03), IEEE Computer Society, Washington, DC, USA, 2003, p. 116.
[71] A. J. Ko, B. A. Myers, Debugging reinvented: Asking and answering why and why not questions about program behavior, in: Proceedings of the 30th International Conference on Software Engineering, ICSE '08, ACM, New York, NY, USA, 2008, pp. 301–310. doi:10.1145/1368088.1368130.
URL http://doi.acm.org/10.1145/1368088.1368130
[72] M. Golan, D. R. Hanson, Duel — a very high-level debugging language., in: USENIX Winter, 1993, pp. 107–118.
[73] E. Bousse, J. Corley, B. Combemale, J. Gray, B. Baudry, Supporting Efficient and Advanced Omniscient Debugging for xDSMLs, in: 8th International Conference on Software Language Engineering (SLE) , Pittsburg, United States, 2015.
URL https://hal.inria.fr/hal-01182517
[74] B. Lee, M. Hirzel, R. Grimm, K. S. McKinley, Debug all your code: Portable mixed-environment debugging, in: Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '09, ACM, New York, NY, USA, 2009, pp. 207–226. doi:10.1145/1640089.1640105.
URL http://doi.acm.org/10.1145/1640089.1640105
[75] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, J. J. LaViola, Jr., Code bubbles: a working set-based interface for code understanding and maintenance, in: CHI '10: Proceedings of the 28th international conference on Human factors in computing systems, ACM, New York, NY, USA, 2010, pp. 2503–2512.

doi:10.1145/1753326.1753706.

[76] A. Zeller, D. Lütkehaus, DDD — a free graphical front-end for Unix debuggers, SIGPLAN Not. 31 (1) (1996) 22–27. doi:10.1145/249094.249108.

[77] J. H. Cross, II, T. D. Hendrix, D. A. Umphress, L. A. Barowski, J. Jain, L. N. Montgomery, Robust generation of dynamic data structure visualizations with multiple interaction approaches, Trans. Comput. Educ. 9 (2) (2009) 13:1–13:32. doi:10.1145/1538234.1538240.
URL http://doi.acm.org/10.1145/1538234.1538240

[78] Y. P. Cheng, J. F. Chen, M. C. Chiu, N. W. Lai, C. C. Tseng, xDIVA: a debugging visualization system with composable visualization metaphors, in: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, OOPSLA Companion '08, ACM, New York, NY, USA, 2008, pp. 807–810. doi:10.1145/1449814.1449869.

[79] M. van den Brand, B. Cornelissen, P. Olivier, J. Vinju, TIDE: A generic debugging framework — tool demonstration —, Electronic Notes in Theoretical Computer Science 141 (4) (2005) 161 – 165, proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications (LDTA 2005) Language Descriptions, Tools, and Applications 2005. doi:10.1016/j.entcs.2005.02.056.
URL http://www.sciencedirect.com/science/article/pii/S1571066105051789

[80] P. R. Henriques, M. J. V. Pereira, M. Mernik, M. Lenic, J. Gray, H. Wu, Automatic generation of language-based tools using the LISA system, Software, IEE Proceedings - 152 (2) (2005) 54–69. doi:10.1049/ip-sen:20041317.
URL http://dx.doi.org/10.1049/ip-sen:20041317

[81] R. E. Faith, L. S. Nyland, J. F. Prins, KHEPERA: a system for rapid implementation of domain specific languages, in: DSL'97: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997, USENIX Association, Berkeley, CA, USA, 1997, pp. 19–19.
URL http://www.cs.unc.edu/~faith/faith-dsl-1997.ps

[82] D. Pavletic, M. Voelter, S. Raza, B. Kolb, T. Kehrer, Extensible debugger framework for extensible languages, in: J. A. de la Puente, T. Vardanega (Eds.), Reliable Software Technologies – Ada–Europe 2015, Vol. 9111 of Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 33–49. doi:10.1007/978-3-319-19584-1_3.
URL http://dx.doi.org/10.1007/978-3-319-19584-1_3

[83] R. Mannadiar, H. Vangheluwe, Debugging in domain-specific modelling, in: B. Malloy, S. Staab, M. van den Brand (Eds.), Software Language Engineering, Vol. 6563 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 276–285. doi:10.1007/978-3-642-19440-5_17.
URL http://dx.doi.org/10.1007/978-3-642-19440-5_17